Facultad de Informática

UNIVERSIDADE DA CORUÑA

TRABAJO FIN DE GRADO
GRADO EN INGENIERÍA INFORMÁTICA
MENCIÓN EN COMPUTACIÓN

Biblioteca para el apoyo al desarrollo de aproximaciones generales en el aprendizaje por refuerzo

Estudiante: Daniel López López
Director: Enrique Fernández Blanco

A Coruña, 25 de junio de 2020.

A mi madre

Agradecimientos

A Patricia por su apoyo incondicional.

Al director de proyecto por su ayuda, consejo y compromiso.

Resumen

Este trabajo aborda la creación de una biblioteca para facilitar la implementación de de modelos de aprendizaje por refuerzo usando un nuevo tipo de aproximación que se basada en la estructura de las partidas de los juegos de mesa.

La biblioteca expone una serie de interfaces para su implementación por parte del desarrollador. Esas interfaces son "Juego", "Jugador", "Ronda", "Tablero" y "Acción". Así cualquier problema que encaje en esas interfaces podrá ser resuelto por un modelo entrenado de esta manera.

Con el fin de validar la biblioteca, así como servir de ejemplo de uso, se ha creado un juego de estrategia que implementa las interfaces listadas anteriormente. Sobre dicho juego se realiza un conjunto de pruebas con el fin de comprobar la utilidad de la biblioteca.

Las pruebas se realizan en torno a un modelo de red neuronal que se entrena primeramente contra el algoritmo de *Monte Carlo Tree Search* muy utilizado en videojuegos para implementar la inteligencia artificial. Tras esto, se pasa a realizar de nuevo un entrenamiento, pero esta vez contra otro modelo similar al que se está entrenando. Por último, se comprueba contra un jugador cómo de bien juega el modelo que se ha entrenado. Se recogen datos de cada una de estas fases y se analizan para valorar el comportamiento de la biblioteca.

Abstract

This paper addresses the creation of a library to facilitate the implementation of reinforment-learning models using a new approach that is based on the structure of board games.

The library exposes a series of interfaces for its implementation by the developer. Those interfaces are "Game", "Player", "Round", "Board" and "Action". So any problem that fits in those interfaces can be solved by a model trained this way.

In order to validate the library, as well as serve as an example of use, a strategy game has been created that implements the interfaces listed above. A set of tests have been performed on this game in order to check the usefulness of the library.

The tests are carried out around a neural network model that is first trained against the *Monte Carlo Tree Search* algorithm widely used in video games to implement artificial intelligence. After this, training is carried out again, but this time against another model similar to the one being trained. Finally, it is checked against a human player how well the model that has been trained plays. Data has been collected from each of these phases and analyzed to assess the behavior of the library.

Palabras clave:

- Aprendizaje automático
- Aprendizaje por refuerzo
- Juego de mesa
- Keras
- Python
- Tensorflow

Keywords:

- Machine Learning
- Reinforcement learning
- Board Game
- Keras
- Python
- Tensorflow

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

Hoy en día uno de los campos más importantes en las ciencias de la computación es el aprendizaje automático (*machine learning* en inglés). El aprendizaje automático estudia el desarrollo de algoritmos que generen modelos basados en un conjunto de datos, los cuales sean capaces de generalizar un comportamiento al ser usados en un conjunto de datos mayor al primero. Con esto se busca tomar decisiones o predecir sobre un nuevo conjunto de datos que sea más grande sin tener que programar un nuevo modelo específicamente para dicho conjunto.

El aprendizaje automático es un campo muy amplio de estudio, el cual está compuesto de diversas aproximaciones o algoritmos. Es por ello que el campo se puede, o suele, subdividir en base a las características de estos algoritmos dando lugar a subcampos dentro del aprendizaje automático, siendo la principal diferenciación entre: algoritmos de aprendizaje supervisado y algoritmos de aprendizaje no supervisado. Estos dos campos son, a su vez, también divisibles en otros campos de estudio más pequeños. Por ejemplo, en el caso del aprendizaje no supervisado encontramos un área conocida como aprendizaje por refuerzo. Si bien, cabe destacar que, a día de hoy, el aprendizaje por refuerzo está ganando su independencia[3] para situarse a la altura del aprendizaje supervisado y el no supervisado. El aprendizaje por refuerzo se basa en el entrenamiento de un modelo sin el uso de un conjunto de datos pares entrada/salida. Dicho aprendizaje basa su funcionamiento en la toma de acciones sobre un entorno, estableciendo distintas recompensas según la acción tomada y utilizando el estado en el que se encuentra tras realizar la acción como retroalimentación.

Los usos para el aprendizaje por refuerzo son muy variados, estos van desde el entrenamiento de vehículos autónomos[4], la robótica[5], la química, el *deep learning*, hasta los juegos[6]. Un ejemplo bastante conocido del uso de este tipo de aprendizaje es AlphaGO

Zero[7]. Esta es la sucesora de AlphaGO una inteligencia artificial que puede jugar al GO al nivel de un jugador profesional, llegando incluso a ganar a algunos de ellos. AlphaGO Zero superó a su predecesora en tan solo dos días gracias al aprendizaje por refuerzo y, además, dio lugar al AlphaZero, otra inteligencia artificial que, generalizando el enfoque de AlphaGo Zero, no solo es capaz de jugar a un nivel por encima de cualquier jugador profesional al GO, sino también al ajedrez y al *shogi*.

A pesar de las evidentes ventajas, el aprendizaje por refuerzo también presenta sus problemas y limitaciones. Uno de los principales problemas y más conocido es el de asignación de puntos. Este problema sucede cada vez que una buena jugada por parte del modelo termina por perder el punto contra su rival, suponiendo una asignación de puntos muy baja o negativa y, por lo tanto, impidiendo al modelo aprender de dicha jugada perdiendo conocimiento útil. Otro problema de perdida de conocimiento por parte del modelo viene dado por la limitación de la gran parte de aplicaciones o técnicas que usan este tipo de aprendizaje al realizar únicamente enfrentamientos cara a cara en el caso de la resolución de juegos. En la mayor parte de los casos el modelo se enfrenta a un único rival, dando lugar a que su aprendizaje se vea solamente influenciado por él, esto puede suponer una perdida de capacidad por parte del modelo al enfrentarse contra un oponente distinto que tome distintas decisiones.

En busca de una solución para este último problema se propone el uso de la estructura de un juego de mesa clásico para la realización de entrenamientos más eficaces. Para ello, se deberá tomar al problema a resolver como el propio juego de mesa, así como las distintas técnicas de resolución del problema como jugadores, ya sean algoritmos o usuarios. Tomando esta estructura se puede conseguir que el modelo aprenda de más de un oponente durante el mismo enfrentamiento, rompiendo una de las limitaciones anteriormente comentadas. Este enfoque, claramente experimental, no dispone en la actualidad de ningún tipo de *software* de soporte o biblioteca que facilite la realización de pruebas y el posterior estudio de los resultados. Por esto, en este Trabajo Fin de Grado, se acomete el desarrollar de una biblioteca que sirva de apoyo para la realización de trabajos que sigan este enfoque.

1.1 Problemática

Aunque están más detallados en el capítulo 2, en este apartado se desarrollan los problemas y limitaciones a los que se enfrenta todo trabajo que usa el aprendizaje por refuerzo. Para ello, se centra en aquellos aspectos que se cree posible evitar con la aproximación a utilizar siguiendo la estructura de un juego de mesa, así como se explica cómo se utiliza dicha estructura.

En la estructura de un juego de mesa básico, una ronda es el conjunto de turnos tomados por los jugadores en la que, como máximo, cada uno de los jugadores realizará uno. En cada turno un jugador realiza un conjuntos de acciones, donde el tamaño posible del conjunto viene limitado por el problema. Realizar el entrenamiento del modelo tras cada ronda permite tener en cuenta cómo ha actuado el rival y ha modificado el estado del tablero.

El problema de asignación de puntos supone una perdida de conocimiento para el modelo, ya que aunque en una ronda del enfrentamiento este haya perdido contra su rival, su decisión puede ser buena a largo plazo. Es decir, que se pierda en una ronda no siempre desemboca en la perdida del enfrentamiento y tampoco significa que se haya realizado un mal movimiento. Para evitar este problema a la hora de asignar recompensas al modelo, es posible realizar la asignación de recompensas tras varias rondas del enfrentamiento y no tras cada una de ellas. De esta forma, si un movimiento por parte del modelo desemboca en una ronda peor que la de su rival pero en varias siguientes mejores, el modelo se verá correctamente recompensado y tendrá en cuenta el movimiento tomado cuando vuelva a encontrarse en una situación similar.

Por otra parte, se tiene la limitación que supone que los enfrentamientos sean únicamente contra un rival, que podría dar lugar a un estancamiento en el modelo y supone realizar más iteraciones contra distintos oponentes si se quiere expandir el conocimiento del mismo. Para esta limitación el concepto de juego de mesa supone la posibilidad de enfrentarse contra más de un rival si el problema lo permite. Gracias al concepto de ronda dentro de los juegos de mesa es posible que cualquier número de jugadores realicen una acción en cada ronda, permitiendo al modelo entrenarse con un entorno en el que el estado varía por las acciones de un número mayor de oponentes. Esto, que se comenta aquí, es completamente compatible con la asignación de recompensas cada varias rondas.

Relacionado con este último punto se puede destacar el hecho de que al existir más de un oponente también es posible entrenar dos modelos a la vez, así como comparar su rendimiento resolviendo el mismo problema y enfrentándolos a otro algoritmo o usuario.

1.2 Objetivos

El objetivo principal es la creación de una biblioteca para el aprendizaje por refuerzo de diferentes técnicas de aprendizaje automática basado en la aproximación mencionada en el apartado 1.1.

Se hace por tanto necesario desglosar este objetivo principal en otros más específicos para

poder alcanzarlo. Los objetivos requeridos para cumplir con el trabajo son los siguientes:

- El primero objetivo es diseñar la biblioteca para que resuelva el problema y, al mismo tiempo, sea fácilmente implementable. Este diseño debe seguir el enfoque y la estructura de un juego de mesa, por lo que deben existir elementos como turno, ronda, jugador y juego. Siendo cada uno se ellos adaptable según el problema en particular.
- El segundo objetivo es implementar dicha biblioteca exponiendo una API clara y amigable. Esta API debe permitir la implementación de un problema como un juego, unos algoritmos resolutorios o modelos como jugadores y que se pueda implementar el concepto de ronda como se considere necesario.
- El tercer objetivo es implementar un ejemplo de uso de la biblioteca, en el que se utiliza un problema a resolver junto con un algoritmo que lo resuelva. En este caso se utiliza un juego como problema a resolver y se desarrolla un algoritmo que lo resuelva, así como una interfaz de usuario simple para que un jugador pueda enfrentarse al modelo.
- El cuarto objetivo es realizar el entrenamiento de uno o varios modelos para que también resuelva el problema usado de ejemplo. Este entrenamiento se realiza contra el algoritmo implementado anteriormente y se pone a prueba contra otros modelos entrenados de la misma forma, contra un jugador o contra el algoritmo usado anteriormente para realizar comparativas en el rendimiento del modelo.
- El quinto objetivo es analizar los resultados de la experimentación con el ejemplo implementado para comprobar la utilidad de la biblioteca desarrollada. Es necesario analizar los resultados obtenidos durante los entrenamientos realizados, así como durante las pruebas. Esto demuestra como de eficaz es la biblioteca implementada y ayuda a escoger posibles trabajos futuros sobre la misma.

1.3 Estructura y contenidos del documento

La memoria se estructura de la siguiente forma:

- **Introducción.** El capítulo 1 de la memoria se dedica a la introducción y explicación de la problemática, así como a la explicación de los objetivos del trabajo.
- **Modelo Conceptual Propuesto.** El capítulo 2 se dedica al estudio de los trabajos ya existentes sobre temas similares al del trabajo y sobre el aporte de este trabajo al mismo.

- **Metodología y planificación inicial.** El capítulo 3 se dedica a a la explicación de la metodología que se usa y de cómo se usa en el desarrollo del proyecto y a su planificación, así como al análisis de los costes de su realización.
- **Desarrollo.** El capítulo 4 explica más en detalle como se organiza y planifica el desarrollo a través de cada uno de los *sprints* que lo forman.
- **Ejemplo de experimentación.** El capítulo 5 se dedica a exponer el desarrollo de la implementación del ejemplo con el que se experimenta el uso de la biblioteca y al análisis de los resultados obtenidos.
- **Conclusiones y futuros desarrollos.** Por último, el capítulo 6 se dedica a expresar las conclusiones a las que se llega tras la realización del trabajo y a explicar posibles vías de desarrollo futuro.

Capítulo 2

Modelo Conceptual Propuesto

El presente capítulo tiene dos objetivos: el primero, realizar una revisión de aquellos trabajos relacionados con la propuesta de este. Así, por ejemplo, se repasaran aquellos trabajos que han adaptado el aprendizaje por refuerzo a una estructura de juego de mesa o similares. El segundo de los objetivos es describir en detalle la propuesta teórica que aquí se contempla, así como destacar las características que lo diferencian de los otros trabajos mencionados anteriormente.

2.1 Revisión bibliográfica

Usado habitualmente en los casos en los que no se tiene un conjunto de ejemplos con los que entrenar, el aprendizaje por refuerzo es aquel que se realiza cuando un modelo aprende mediante prueba y error mediante una evaluación sobre los resultados de las acciones que este toma.

Una de las personas más conocidas por su trabajo en este campo es Richard S. Sutton[8], considerado uno de los padres del aprendizaje por refuerzo. Durante los años 80 trabajó en lo que fueron las bases que han dado lugar al aprendizaje por refuerzo que se utiliza hoy en día.

Como ya se ha comentado en la introducción, el aprendizaje por refuerzo es usado en muchos y diversos campos, como la robótica[9] o las finanzas[10]. Aunque el campo en el que más ha destacado en los últimos años es en la resolución de juegos de mesa, llegando a demostrar habilidades sobrehumanas.

El caso más conocido en el que se ha usado el aprendizaje de refuerzo para resolver un

juego de mesa es, tal vez, AlphaGo[11]. En este trabajo se consiguió superar el nivel de jugadores profesionales de GO, siendo este, hasta ese momento, considerado uno de los juegos de mesa más difíciles para la inteligencia artificial. AlphaGO está basado en una red neuronal que se entrenó primero usando aprendizaje supervisado utilizando a jugadores expertos y después con aprendizaje por refuerzo realizando enfrentamientos contra si mismo. En esta segunda fase consiguió una precisión casi equivalente a algoritmos como el árbol de búsqueda de Monte Carlo[12], necesitando mucho menos tiempo que este. En la figura 2.1 se puede ver el nivel en Elo[1] en comparativa con el jugador profesional Fan Hui.

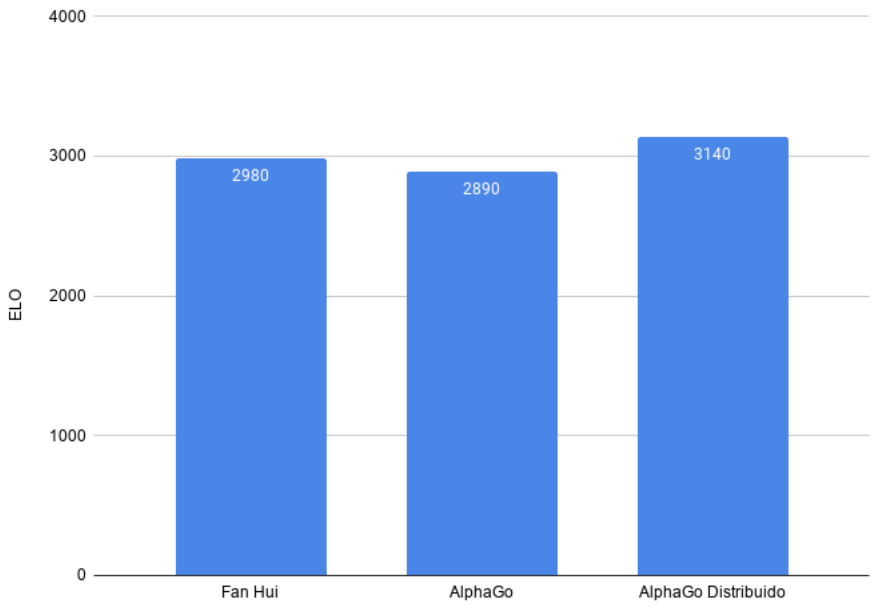

Figura 2.1: Comparativa de Elo entre AlphaGo y el jugador profesional al que venció 5 a 0.

Tras AlphaGo se creó un nuevo programa llamado AlphaGo Zero[13]. En esta ocasión se superó en unos pocos días a su predecesor y usando únicamente aprendizaje por refuerzo para entrenar su única red neuronal. Es interesante destacar que simplificaron los datos de entrada a tan solo las piezas en tablero. El avance de capacidades a la hora de jugar al Go se puede ver claramente en la figura 2.2 representado por el Elo de cada programa.

[1]Método matemático para calcular la habilidad relativa de los jugadores en juegos como el Go.

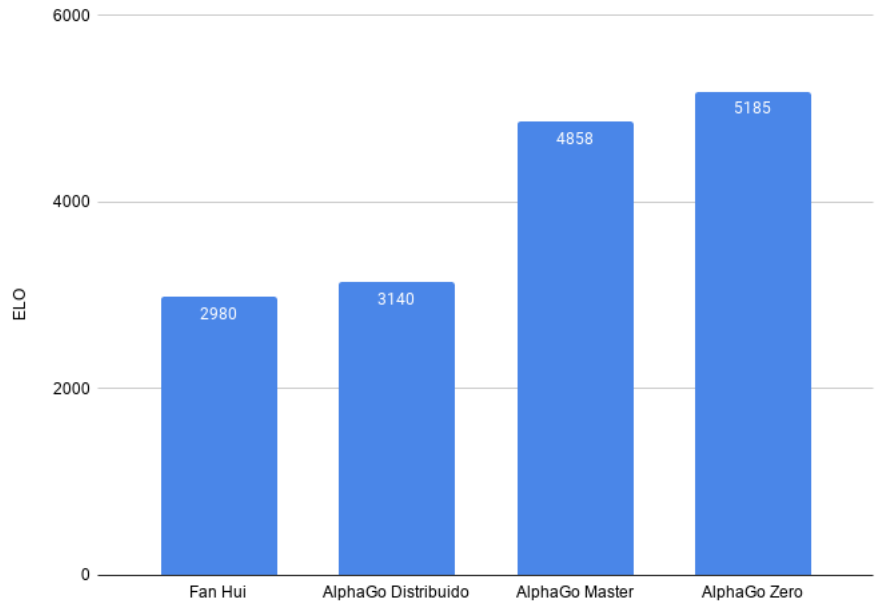

Figura 2.2: Comparativa de entre AlphaGo Zero y sus predecesores.

La generalización del programa anterior dio lugar a AlphaZero[1]. Este programa superó en solo un día a AlphaGo Zero y, además, supero el nivel humano también en ajedrez y *shogi*. Esto demuestra que tan solo entrenando con aprendizaje por refuerzo y sin conocimientos previos es posible conseguir un algoritmo con un rendimiento súper humano en varios juegos de mesa complejos. En la figura 2.3 se ve como supero en habilidad a cada una de las otras soluciones destacables tanto en ajedrez como en *shogi* y Go.

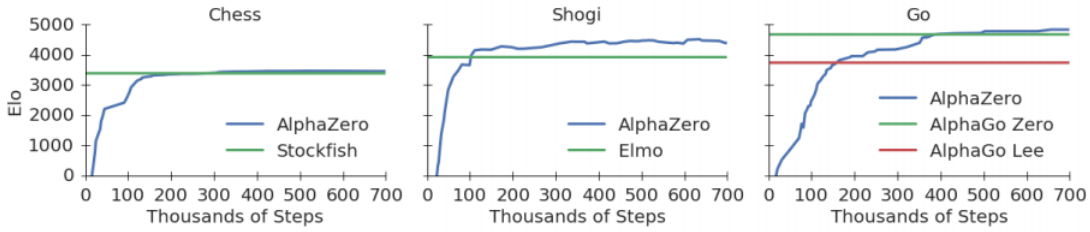

Figura 2.3: Comparativa de entre AlphaZero y las otras inteligencias artificiales destacables en cada juego. Fuente: *A general reinforcement learning algorithm that masters chess, shogi and Go through self-play*[1]

Esta no es la primera vez que los juegos de mesa se ven relacionados con el aprendizaje por

refuerzo. En el trabajo de Imran Ghory con título *Reinforcement learning in board games*[14], se trata el uso de aprendizaje por diferencia temporal para el desarrollo de un agente capaz de jugar a juegos de mesa, centrándose en cómo obtener datos para realizar el entrenamiento con distintos métodos y en las características claves de un juego de mesa que pueden afectar a dicho aprendizaje. El aprendizaje por diferencia temporal[15] es un tipo de algoritmo de aprendizaje por refuerzo que no usa modelo y aprende por prueba y error. Aunque en este trabajo solamente se tienen en cuenta los juegos de mesa en los que como máximo dos jugadores se enfrentan.

Este trabajo busca proveer de una biblioteca que ayude a aplicar este tipo de algoritmos en casi cualquier tipo de problema, manteniendo esta aproximación en el entrenamiento. Como se puede comprobar, programas como el AlphaZero son brillantes resolviendo juegos de mesa, por lo que se intenta generalizar esto a otro tipos de problemas.

2.2 Aportaciones del trabajo propuesto

En este trabajo se propone realizar el caso inverso a lo que se acaba de mencionar, de esta forma es la estructura de los juegos de mesa lo que modificará al aprendizaje por refuerzo. Para ello se han buscado los puntos en común entre todos los juegos de mesa, intentando generalizarlo lo máximo posible. Si se analizan dichos puntos se puede llegar a que todo juego de mesa está formado en su núcleo por unos pocos elementos básicos:

- **Juego**. Es el conjuntos de reglas y objetivos que indican cómo se juega y para qué. Además se podría decir que dentro del juego podemos englobar los siguientes elementos:
  - Tablero. Es el elemento que define la situación en un momento determinado del juego.
  - Puntuación. Es la suma de los puntos acumulados por cada uno de los jugadores que participan en la partida.
  - Reglas que definen las posibles acciones. Son el conjunto de reglas que se aplican en un momento determinado del juego para acotar las posibles acciones a realizar en dicho momento. Por ejemplo, en el caso del ajedrez en caso de un jaque las reglas limitan al jugador a realizar una acción que evite dicha situación.
- **Acción**. Es aquello que cambia el tablero del juego y, potencialmente, su puntuación.
- **Jugador**. Es aquel que realiza las acciones y al que se le asignan los puntos dentro del juego.

- **Turno**. Es el tiempo en el que un jugador realiza sus acciones. Cada jugador tiene su turno y a cada turno lo sigue otro hasta que la partida finaliza.
- **Ronda**. Es el conjunto de los turnos de todos los jugadores, con un solo turno por jugador. Es decir, cuando un jugador vuelve a estar en su turno la ronda habrá cambiado.
- **Partida**. Es el conjunto de rondas necesarias para terminar el juego. Normalmente la puntuación en este punto es la que decide que jugador gana.

Por otro lado, tenemos el aprendizaje por refuerzo que tiene los siguientes elementos básicos:

- **Entorno**. Es aquello sobre lo que el agente toma acciones.
- **Agente**. Es quién, observando el estado en un momento dado del entorno, realiza una acción.
- **Acción**. Es aquello que cambia el estado en el que se encuentra el entorno para que este pase a un nuevo estado.
- **Recompensa**. Es lo qué el agente recibe para conocer si su acción a resultado en un nuevo estado favorable del entorno o no.
- **Estado**. Es cada una de las situaciones por las que pasa el entorno.
- **Interprete**. Es quién observando el nuevo estado del entorno decide la recompensa que se le asigna al agente.

Habitualmente en el aprendizaje por refuerzo el flujo sigue una estructura en la que el agente observa el estado en el que se encuentra el entorno, decide realizar una acción, esta modifica el entorno haciendo que transicione a un nuevo estado, el interprete observa dicho estado y decide que recompensa debe recibir el agente. El agente vuelve a comenzar el flujo recibiendo el nuevo estado del entorno y, esta vez, una recompensa que le permite aprender sobre la última acción que realizó.

Una vez analizados los puntos básicos de cada parte, es posible buscar cómo aplicar los elementos de un juego de mesa sobre los elementos del aprendizaje por refuerzo. El elemento sobre el que se aplican las acciones en el caso de un juego de mesa es sobre el propio juego, mientras que en el aprendizaje por refuerzo es el entorno, de esto se puede deducir que ambos tienen un rol común. También, gracias a las acciones se puede identificar que el agente que

toma acciones sería un jugador en el caso del juego de mesa. Además, los estados generados por dichas acciones serían equivalentes a los elementos de tablero, puntuación y reglas, ya que estos definen la situación en la que se encuentra un juego, es decir su estado.

Por otro lado, tanto recompensa como interprete son elementos propios del aprendizaje por refuerzo que no se ven representado de algún modo por ningún elemento del juego de mesa. Aún así, podría decirse que el flujo que sigue el aprendizaje por refuerzo sería equivalente a un turno o a una ronda, cuando más de un agente interactúan con el entorno. Normalmente el agente recibe una recompensa cada vez que modifica el estado del entorno, pero si usamos el concepto de ronda sería posible tomar la decisión sobre la recompensa cada una o más rondas.

Por lo tanto, el objetivo es permitir que cualquier problema que pueda ser tratado como un juego de mesa pueda ser resuelto por un modelo entrenado por aprendizaje por refuerzo. Para ello dicho problema será el juego y los algoritmos que lo resuelven serán los jugadores.

En el apartado Problemática del primer capítulo ya se comentó que esta forma de realizar entrenamientos siguiendo el aprendizaje por refuerzo, tiene unas cuantas ventajas sobre la estructura básica:

- Con el concepto que ofrece el elemento ronda de un juego de mesa es fácilmente implementable que el entrenamiento del modelo se realice una vez hayan pasado un número determinado de rondas. Esto permite experimentar hasta obtener el mejor resultado. Normalmente el entrenamiento se realiza por cada flujo de toma de acciones que modifican el estado del entorno por parte del modelo. Por ejemplo, si el modelo realiza una acción que supone una perdida inmediata pero una ganancia a largo plazo las recompensas que recibiría en ese momento serían negativas o nulas, pero si la recompensa y posterior entrenamiento se realiza cada varias rondas se podría tener en cuenta que tras cierto número de rondas ha habido una ganancia para el modelo. El número correcto de rondas ha esperar entre recompensas y entrenamientos se obtiene experimentando, ya que un número muy bajo podría no tener en cuenta acciones realizadas buscando beneficio a largo plazo, pero un número demasiado alto sería casi como tener en cuenta solo el resultado final ignorando acciones que hayan significado ganancias aunque fuesen pequeñas.
- Muchos juegos de mesa solo enfrentan a dos jugadores, pero es cierto que es posible que los juegos permitan un mayor número de participantes. Por esto es posible entrenar un modelo contra varios rivales para que aprenda de todos ellos. Es decir, un modelo podría aprender tanto de un algoritmo que resuelve el problema, como de un experto

humano enfrentándose a ambos a la vez.

- Como continuación del punto anterior se podría añadir un nuevo modelo a la partida y, de esta forma, entrenar dos modelos enfrentándose entre si, pero también contra el algoritmo y el experto. Como el estado en el que recibiría el entorno cada modelo sería distinto su entrenamiento daría lugar a modelos distintos.
- Igualmente, es posible validar más de un modelo, o enfrentarlos y observar cuál ofrece mejores resultados. De esta forma, si se cogen dos modelos ya entrenados es posible, manteniendo la estructura de juego de mesa, enfrentarlos para comparar los resultados que obtiene cada uno resolviendo el mismo problema.

Para una mayor claridad se listan los beneficios esperados que supone todo lo aquí propuesto:

- Evita la perdida de conocimiento a largo plazo.
- Permite aprender de más de un oponente durante cada entrenamiento.
- Da la oportunidad de entrenar más de un modelo a la vez.
- Posibilita la comparación de dos o más modelos sobre el mismo problema.

Capítulo 3

Metodología y Planificación Inicial

Debido al tipo de proyecto a afrontar, se hace necesario buscar una metodología que permita un estilo de desarrollo ágil. Se necesita que, además de ágil, sea incremental, ya que una vez se tenga una versión preliminar de la API sufrirá, muy seguramente, cambios al ir realizando las pruebas sobre un ejemplo.

Por lo comentado se ha decidido trabajar cogiendo como base la metodología Scrum [16] y utilizando un tablero Kanban [17] para el control de las tareas.

3.1 Metodología

3.1.1 Scrum

Sabiendo que la mejor forma de conseguir una API funcional y robusta es realizarla de forma incremental, la metodología escogida debe adaptarse a ello. Como se ha indicado anteriormente, se ha optado por el uso de Scrum por dos motivos principalmente:

- Permitirá reaccionar a los cambios en los requerimientos de la API, que surgirán a lo largo de la realización de las pruebas sobre la misma.
- Lograr una mayor productividad que con el uso de metodologías más clásicas, debido a la eliminación de gran parte de la burocracia que se realiza en estas últimas.

Scrum se basa en tres pilares: equipo, eventos y artefactos. A continuación se hablará en mayor detalle de cada uno de ellos.

Equipo

Un equipo Scrum está formado por tres perfiles: *Scrum Master*, *Product Owner* y desarrollador, este último es un perfil realizado por más de una persona, mientras que los dos primeros son, normalmente, tomados por un único individuo.

El ***Scrum Master*** es el encargado de la aplicación de Scrum durante todo el desarrollo del proyecto. Está al servicio del equipo de desarrollo, de forma que ninguna otra persona pueda interponerse en su trabajo. Ayuda a mediar entre el *Product Owner* y el equipo, explicando como ordenar y gestionar el *Product Backlog* al *Product owner* y los objetivos, alcance y dominio del producto a los miembros del equipo de desarrollo. Además, debe encargarse de organizar los eventos de Scrum de un modo que todas las partes puedan acudir a los mismos.

El ***Product Owner*** se encarga de conseguir un producto con el mayor valor posible. Es el último responsable del *Product Backlog*, por lo que es el encargado de crear, actualizar y mantener los elementos que lo conforman. También debe priorizar dichos elementos, siendo esta priorización la que debe seguir el equipo de desarrollo. Es importante que sigan las directrices marcadas por él, para mantener útil este perfil.

Por último, el **equipo de desarrollo** es el encargado de transformar los elementos del *Product Backlog* en funcionalidades mediante la creación del producto en incrementos. La responsabilidad es repartida entre todos los miembros del equipo, así como la capacidad de auto-gestionarse, organizarse y asignarse el trabajo.

Eventos

Scrum define y necesita un conjunto de eventos para su funcionamiento. Estos eventos, en su mayor parte reuniones, están pensados para aumentar la transparencia y la inspección. Cada uno de ellos tiene un objetivo que se debe tratar de cumplir dentro de sus limites.

El ***Sprint*** es el evento base que engloba los demás eventos (Figura 3.1). Tiene como objetivo la creación de un incremental del producto, que se espera que sea ejecutable. El conjunto de los *sprints* forma el total del desarrollo. Deben tener una duración fija que debería estar entre 1 y 4 semanas. Durante su duración los objetivos y requisitos sobre el producto no pueden verse modificados, por este motivo se recomienda la duración comentada anteriormente. La cancelación de un *sprint* debería de suceder solo en último caso y, si así ocurriese, se debería intentar aceptar todos los elementos del *Product Backlog* que han sido terminados durante su duración.

Dentro del *sprint* están los eventos que son reuniones, cada uno de ellos con una duración y objetivos distintos.

La **reunión diaria** es un evento que, como su nombre indica, se realiza a diario durante todo el *sprint*, reuniendo a todo el equipo de desarrollo y al *Scrum Master* a la misma hora y el mismo lugar cada día durante un máximo de 15 minutos. En esta reunión se discuste sobre el trabajo que se ha realizado el día anterior y el que se realizará el propio día, así como se comenta cualquier posible bloqueo. El responsable de la realización de esta reunión es el *Scrum Master*, aunque los que deben actuar de forma activa son los miembros del equipo.

Al inicio de cada *sprint* se realiza la reunión de **planificación de *sprint***, en la cual participa todo el equipo Scrum. Tiene una duración de 8 horas máximo, si los *sprints* durasen 1 mes, si no la duración debería ser menor. Durante esta reunión el *Product Owner* decide el objetivo del *sprint* y que elementos del *Product Backlog* pasan a formar parte del *Sprint Backlog* para lograr dicho objetivo. Los miembros del equipo deciden que tareas son necesarias para la realización de cada uno de dichos elementos, que también pasan a formar parte del *Sprint Backlog*. La responsabilidad de la realización de esta reunión recae también en el *Scrum Master* y todo el equipo Scrum debe participar de forma activa.

Por otra parte, al final de cada *sprint* se organiza una reunión para la **revisión del *sprint***. En este caso la duración debería ser menor a las 4 horas. Como objetivo tiene la discusión sobre la evolución del *sprint* por parte del equipo de desarrollo, así como la exposición del incremento resultante al *Product Owner* y a cualquier persona que él considere. Además, se realizará una revisión del *Product Backlog* para marcar como completados aquellos realizados durante el *sprint* y, si fuese necesario, adaptar los elementos restantes. Como en el caso de las demás reuniones, es resposabilidad del *Scrum Master* que se realice y debe acudir todo el equipo Scrum y cualquier persona que el *Product Owner* considere necesaria.

Como último evento, que también debe realizarse al final de cada *sprint*, está la reunión de **retrospectiva del *sprint***. Con una duración de menos de 3 horas, en ella se busca identificar posibles puntos en los que el equipo puede mejorar; para ello se inspecciona el desarrollo del *sprint* y se busca que elementos salieron bien, cuales mal y cuales son mejorables. Con dicha información se crea un plan de mejora que el equipo debe implementar cuanto antes. Esta reunión no siempre se realiza, aunque es altamente recomendable su realización y deben acudir y participar tanto el equipo de desarrollo como el *Scrum Master*.

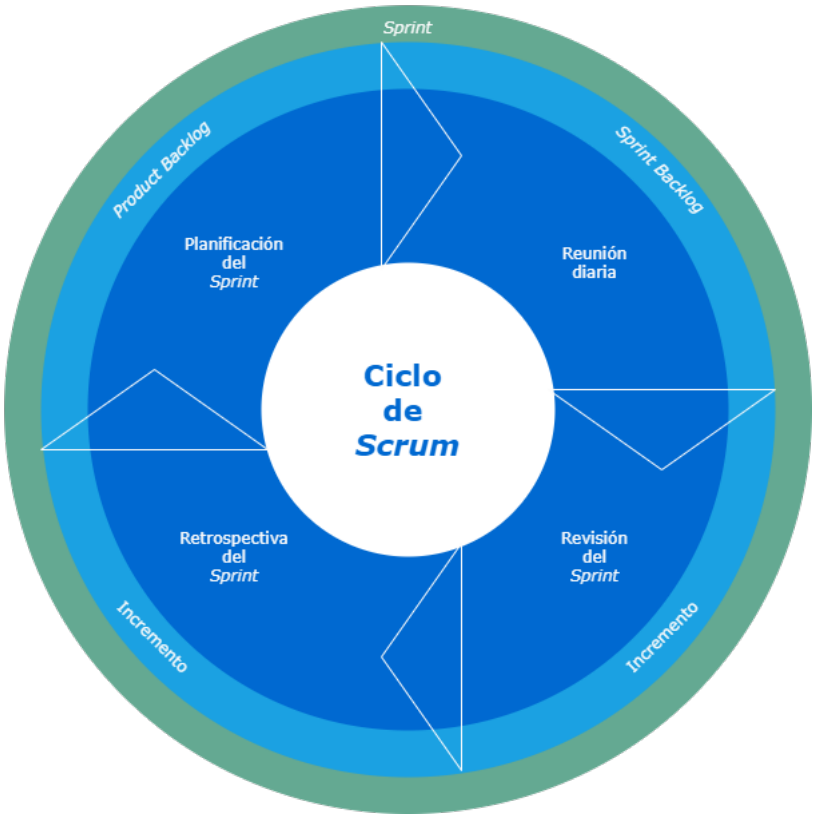

Figura 3.1: Ciclo de *Scrum* de un *sprint*.

Artefactos

Ya nombrados anteriormente durante la explicación de los eventos y el equipo de Scrum, aquí se explicarán los artefactos que se utilizan para aplicar Scrum. Para que estos sean utiles todos los miembros del equipo deben entenderlos de la misma forma, con la ayuda del *Scrum Master*.

Durante todo el desarrollo el ***Product Backlog*** va evolucionando, así como lo hace el producto. Este artefacto es una lista ordenada de lo que es necesario en el producto, por eso a lo largo de los *sprints* se adapta para mantener el producto adecuado, útil y competitivo. Los elementos que lo conforman tienen como propiedades una descripción de qué es necesario, una posición en la prioridad, un valor que aporta, una estimación sobre el trabajo necesario para realizarlo y, de forma no obligatoria pero sí recomendada, una descripción de unas pruebas que deberían superarse para decir que está terminado el elemento. El mayor responsable de este artefacto es el *Product Owner*, aunque todo el equipo debe trabajar con él.

Del *Product Backlog* se extraen elementos al inicio de cada uno de los *sprints* que terminan formando el otro artefacto definido por Scrum, el ***Sprint Backlog***. Este artefacto está compuesto por los nombrados elementos del *Product Backlog*, así como por las tareas creadas por el equipo durante la planificación que son necesarios para cumplir el objetivos del *sprint*. Igual que el *Product Backlog*, evoluciona a lo largo del *sprint* teniendo en cuenta que no es posible añadir más elementos del *Product Backlog* durante la duración del *Sprint Backlog*. Todo el equipo trabaja sobre este artefacto, además es usado durante las reuniones diarias para ver el trabajo restante que queda por realizar durante el *sprint*.

3.1.2 Kanban

Como se ha comentado anteriormente, dentro de cada *sprint* se realizan un número determinado de tareas y para la gestión de estas el método Kanban es de utilidad. Esto es gracias a su enfoque en el proceso de trabajo que se refleja en sus 5 prácticas centrales, que son las siguientes:

- Visualizar el flujo de trabajo y comprenderlo. Esta es una práctica importante que se puede realizar gracias al tablero del que se hablará en el siguiente apartado.
- Limitar la carga de trabajo. Se debe acortar el número de trabajos en proceso, evitando así perder el foco.
- Gestionar el tiempo. Se busca un flujo rápido del trabajo generando valor de forma continuada.
- Ayudar a la visibilidad. Es importante que los miembros del equipo entiendan todo el proceso, por ello debe estar bien definido.
- Usar modelos para lograr una mejora continua. El equipo debe tener un entendimiento común de los problemas a los que se enfrentar para poder proponer acciones de mejora.

Estas prácticas buscan que durante el desarrollo se consigan cumplir los 4 principios de Kanban:

- Comenzar con lo que ya se hace. Kanban no requiere modificar el proceso existente, sino aplicarlo sobre él. De esta forma se identifican y mejoran los posible problemas de una forma gradual.

- Aceptar la búsqueda de cambios incrementales y evolutivos. Es preferible realizar pequeños cambios de forma continua, que provocarán menor resistencia entre el equipo.
- Respetar la organización actual. Es importante conservar aquello que aporta valor y solo realizar cambios en eso que si lo requiere, siempre de forma incremental.
- Fomentar el liderazgo en todos los niveles. Es importante que todo el equipo tenga una mentalidad de mejora continua.

Tablero

El tablero Kanban es un artefacto que permite cumplir de forma fácil muchas de las prácticas nombradas en el anterior apartado. En él es muy cómodo que elementos de trabajo tiene el equipo y en qué estado se encuentran, así como quién está asignado a cada tarea; facilitando eso se ayuda a visualizar el flujo de trabajo, así como a gestionar el tiempo y a limitar la carga.

El flujo de trabajo que muestra el tablero básico se divide en tres pasos o columnas: *to do* (por hacer), *doing* (en curso) y *done* (finalizado). Estos pasos se pueden ampliar con pasos intermedios, aunque debería evitarse añadir demasiados para no complicar la visualización de dicho flujo (Véase Figura 3.2).

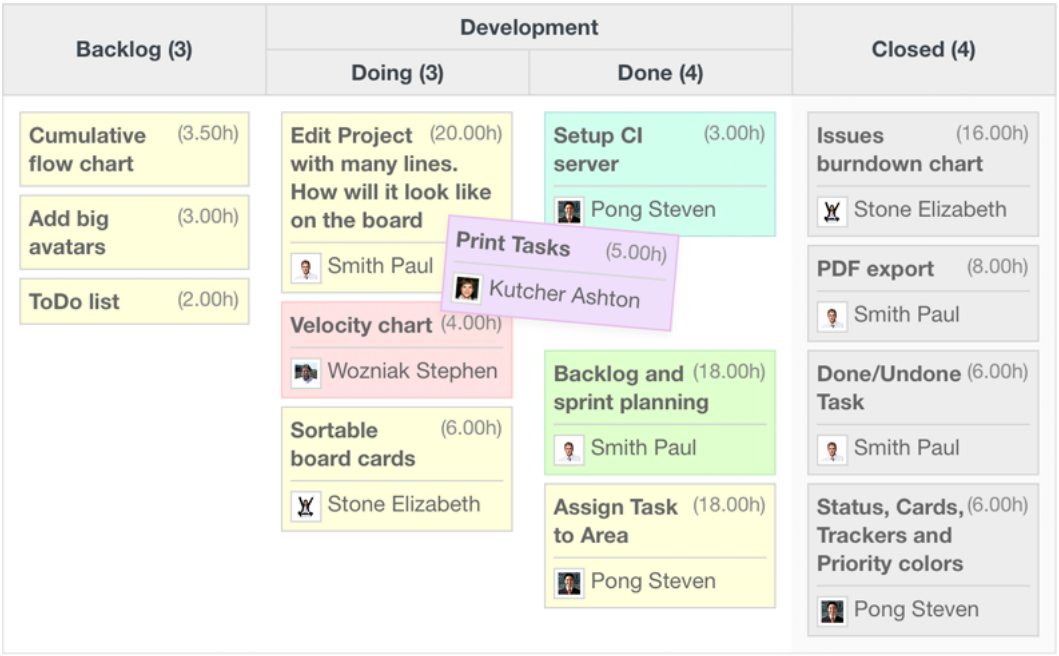

Figura 3.2: Tablero Kanban. Fuente: RedmineUp Agile Plugin[2]

En cada una de las columnas que forman el tablero se encuentran unas tarjetas que deben indicar una pequeña descripción del trabajo a modo de titulo, así como quién se encarga de realizarlo, cuál es la duración prevista y su prioridad .

3.2 Adaptaciones

En este proyecto se ha hecho una adaptación, con fines educativos, de los dos métodos de trabajo comentados en el apartado anterior. Se ha utilizado como metodología principal Scrum añadiendo Kanban mediante el uso de su tablero para la organización del flujo de trabajo, así como usadas sus prácticas para la mejora del mismo.

El mayor problema a la hora de realizar la adaptación para el uso de estos métodos viene dado por ser una única persona en el equipo del proyecto. Algunos eventos de Scrum pierden casi todo su sentido a la hora de ser realizados por una sola persona y algunas de las prácticas de Kanban también tienen menor sentido de uso en este caso.

Los perfiles de Scrum se toman en gran parte por una misma persona, exceptuando parte de las responsabilidades del *Product Owner* que las realiza el director del trabajo. En el caso de los eventos, se sustituye la reunión diaria por un análisis diario del estado de los elementos a realizar durante el *sprint* en curso. También se sustituye la retrospectiva del *sprint* por un análisis continuo de los posibles errores cometidos durante el desarrollo. Por otra parte, se mantienen las reuniones de planificación y revisión del sprint con normalidad, salvo por la duración que es menor a la debida, una hora para cada reunión. Por supuesto tanto *Product Backlog* como *Sprint Backlog* se utilizan de forma normal, aunque la responsabilidad de ambos es repartida entre *Product Owner* y *Scrum Master*/equipo de desarrollo.

Para apoyar el *Sprint Backlog* es usado el tablero Kanban, que tiene la estructura básica de la que ya se ha hablado: *to do*, *doing* y *done*. Los elementos de dicho tablero son las tareas definidas para completar cada uno de los elementos que forman el *Sprint Backlog*. Como dato especifico, se limita el número de tareas que pueden estar en el estado *doing* a solo 2 con el fin de evitar perder el foco del trabajo que se está realizando.

3.3 Toma de requisitos inicial

Como en todo proyecto de ingeniería, el primer paso debe de ser el entender qué se quiere hacer a grandes rasgos y el cómo. Esto nos lleva al concepto de requisito, en concreto, los

proyectos de informática identifican dos tipos, funcionales (RF) y no funcionales (RNF). Los primeros, los funcionales, hacen referencia a los elementos que el usuario o *Product Owner* desea ver reflejados en la versión final de proyecto con el fin de cubrir sus necesidades. Por otro parte, los no funcionales, no hacen hincapié en el qué sino en el cómo ya que marcarán requisitos para que, no sólo el sistema haga la funcionalidad que se espera, sino dentro de un tiempo y recursos adecuados a la casuística.

Por ejemplo, dentro del marco de este trabajo, un requisito funcional sería el poder definir un modelo con el fin de ser entrenado, mientras que un requisito no funcional sería que el sistema pueda ser ejecutado en cualquier arquitectura.

Dicho esto los requisitos iniciales del sistema quedarían como:

- Requisitos funcionales

RF1 - La biblioteca expondrá una interfaz "Jugador" que permitirá la implementación de los distintos métodos que resolverán el problema. Esta interfaz tendrá los métodos necesarios para jugar.

RF2 - La biblioteca expondrá una interfaz "Juego" que servirá para implementar el problema a resolver. El Juego tendrá todos los datos del estado del problema dado un momento y dispondrá de métodos que permitirán realizar las acciones necesarias para modificar dicho estado.

RF3 - La biblioteca expondrá una interfaz "Ronda" que se utilizará para la implementación del concepto ronda según el problema que se quiera resolver. Esta será simple permitiendo una mayor especialización para cada problema.

RF4 - El ejemplo tendrá un sistema que registre todos los datos necesarios de las ejecuciones realizadas para su posterior análisis. Tras cada enfrentamiento se registrarán los datos del mismo en un fichero para, al finalizar, procesarlo y obtener estadísticas.

RF5 - El ejemplo dispondrá de una interfaz de usuario para permitir resolver el problema sin necesidad de usar un algoritmo. Una de las implementaciones de la interfaz jugador deberá ser una interfaz de usuario, para poder probar el modelo contra un jugador real y, si fuese necesario, entrenarlo.

RF6 - El ejemplo permitirá la configuración de diversos parámetros como, por ejemplo, la configuración de los jugadores participantes en el enfrentamiento. Se deberá poder configurar no solo los jugadores, sino el número de rondas a ejecutar, cada cuántas rondas se realiza un entrenamiento y el estado inicial del problema en cada ejecución.

- Requisitos no funcionales

RNF1 - Las interfaces que se expongan de la biblioteca deberán ser fáciles de implementar y expandir para adaptarse a cualquier problema sin requerir un gran esfuerzo.

RNF2 - La biblioteca deberá evitar tener dependencias externas para facilitar su uso.

RNF3 - La biblioteca deberá ser fácilmente ampliable para facilitar trabajos futuros.

RNF4 - El ejemplo implementará todas las interfaces desarrolladas para la biblioteca.

RNF5 - El problema de ejemplo deberá tener cierta complejidad para conseguir unos resultados con datos de cierto interés.

3.4 Planificación inicial

Para la elaboración de este trabajo se ha decidido realizar un total de 6 *sprints*, estos tienen una duración de 2 semanas lo que permite abordar más de un elemento de cierta complejidad en cada uno de ellos.

Se divide el proceso de desarrollo del trabajo en 5 fases que son las siguientes:

- Análisis del problema y requisitos. En esta fase se analizará el problema a resolver y de este análisis se obtendrán los requisitos que debe cumplir el software que se programará tras estas primeras fases.
- Diseño y Arquitectura de alto nivel. Durante esta fase se definirá como deberá funcionar la biblioteca y cómo deberá ser para cumplir los requisitos establecidos.
- Programación de la API. A lo largo de esta fase se realizará el trabajo de programación de la biblioteca, que aplica todo lo diseñado anteriormente.
- Programación del ejemplo. En esta fase se desarrollará un ejemplo de experimentación usando la API programada en la fase anterior que, posteriormente nos ayudará a probar el buen y correcto funcionamiento de la misma.
- Experimentación o pruebas. Durante esta fase se utilizará el ejemplo para probar la API, analizando los resultados que se consigan durante su uso.

En la Tabla 3.1 se muestra como se asignan estas 5 fases principales a cada uno de los 6 *sprints* que forman el total del desarrollo:

Sprint	Fases a realizar
1	Análisis del problema y requisitos Diseño y Arquitectura de alto nivel
2	Programación de la API
3	Programación de la API
4	Programación del ejemplo
5	Programación del ejemplo
6	Experimentación o pruebas

Tabla 3.1: Fases a realizar en cada *sprint*

3.5 Costes

Con el fin de realizar una estimación de los costes separaremos la misma en una primera parte con los costes materiales y otra con los costes de los recursos humanos utilizados durante todo el desarrollo.

Los costes materiales se dividen en los costes dados por los dispositivos así como las herramientas *software* utilizadas, los cuales se listan en la tabla 3.2, y los costes indirectos, como son el alquiler de un estudio, la luz, conexión a internet y agua, indicados en la tabla 3.3. Los costes materiales son prorrateados teniendo en cuenta la estimación de vida útil de cada uno de los bienes y servicios, aunque el proyecto tiene una duración de tan solo 3 meses este prorrateo se realiza como si los materiales fuesen a utilizarse durante toda su vida.

Recurso	Precio(en euros)
Ordenador Sobremesa	200
Ordenador Portátil	75
Licencia Windows	25
Licencia Jira *Standard*	21

Tabla 3.2: Coste de los recursos materiales

Recurso	Precio(en euros)
Alquiler estudio	1800
Luz	150
Conexión internet	120
Agua	30

Tabla 3.3: Costes indirectos

Sumando un total de coste material de **2.421 €**.

En el caso de los costes de recursos humanos se debe tener en cuenta que durante todo el desarrollo, aunque en este caso en particular al tratarse de un trabajo con fines académicos solo ha trabajo una persona, se realizan tareas con dos perfiles distintos. Un perfil de analista que se encarga de tareas de análisis del problema y de los resultados, así como de diseño de la API; y otro perfil de desarrollador encargado de tareas de programación de la API y el ejemplo de experimentación. En la tabla 3.4 se puede comprobar el coste por cada perfil; en este caso, aunque el coste oficial es menor, se ha tomado como referencia la escala salarial de la empresa actual en la que trabaja el alumno.

Recurso	Horas	Precio/Hora(en euros)	Precio
Analista	160	15,625	2.500
Desarrollador	320	12,5	4.000

Tabla 3.4: Costes de recursos humanos

Sumando un total de coste de recursos humanos de **6.500 €**.

Esto suma un coste estimado total de **8.921 €** para la realización de este proyecto.

Capítulo 4

Desarrollo

Tras seleccionar la metodología a utilizar durante el desarrollo, el siguiente paso es planificarlo. Gracias a Scrum, los primeros pasos se pueden resumir en la creación de un calendario de *sprints* y de los requisitos iniciales.

El calendario de *sprints* indica cómo se reparten en el tiempo los mismos y cuándo se realizan las reuniones de planificación y revisión.

Los requisitos iniciales obtenidos para este proyecto se listan en el apartado Toma de requisitos inicial.

El proyecto dispone de tres meses de duración, por lo que se decide asignar una duración de dos semanas por *sprint*. Además, está dentro del rango de duración recomendado por Scrum, de 1 a 4 semanas. Es una duración que, en un trabajo con una extensión total tan corta, hace fácil reaccionar a los posibles cambios necesarios tras recibir los comentarios del *product owner*.

Durante las reuniones de planificación de cada *sprint* se definen las tareas que se realizan en el *sprint* correspondiente, así como el diseño a alto nivel de la implementación de dichas tareas. Por otra parte, en las reuniones de revisión de cada *sprint* se revisa lo que se ha desarrollado durante dicho *sprint* con el fin de determinar si el trabajo realizado cumple o no con los requisitos. En caso de no cumplirlos es necesario indicar los cambios que se desean realizar y adaptar el *product backlog* a dichos cambios.

4.1 *Sprints*

4.1.1 Historias de usuario

Como ocurre normalmente en todo proyecto que utilice una metodología ágil, se desarrolla una lista de historias de usuario que pasarán a explicar de una forma más concisa las características que se le piden al trabajo. Las historias de usuario son pequeñas descripciones que, de una forma simple, explican una característica desde el punto de vista de quién quiere dicha característica. Por esto, se decide analizar y dividir los requisitos iniciales para obtener una lista completa de historias de usuario que realizar a lo largo del desarrollo. Esta lista no es estática, puede cambiar a lo largo del transcurso del proyecto.

Requisito	Historias de usuario obtenidas
RF1	HU1 - Como desarrollador quiero crear clases que puedan actuar como jugadores para poder interactuar con el juego. HU2 - Como creador de la biblioteca quiero probar la interfaz de jugador para comprobar su correcta adaptación.
RF2	HU3 - Como desarrollador quiero crear clases que definan las reglas de un juego para resolver dicho juego. HU4 - Como creador de la biblioteca quiero probar la interfaz de juego para comprobar su correcta adaptación.
RF3	HU5 - Como desarrollador quiero crear clases que definan una ronda para agrupar los turnos de un juego. HU6 - Como creador de la biblioteca quiero probar la interfaz de ronda para comprobar su correcta adaptación.
RF4	HU7 - Como creador de la biblioteca quiero poder visualizar los datos relacionados con un enfrentamiento para su posterior análisis.
RF5	HU8 - Como creador de la biblioteca quiero permitir que un jugador humano interactúe con el juego para la realización de entrenamientos y pruebas. HU9 - Como jugador quiero poder ver el tablero y tomar acciones sobre el juego para jugar.
RF6	HU10 - Como creador de la biblioteca quiero configurar los parámetros que definen un enfrentamiento para realizar distintas pruebas de una forma más cómoda.
RNF1	HU11 - Como desarrollador quiero implementar sobre cualquier problema las interfaces para aprovechar los beneficios de las mismas.

RNF2	HU12 - Como desarrollador quiero utilizar la biblioteca sin necesidad de otras dependencias para evitar posibles problemas.
RNF3	HU13 - Como creador de la biblioteca quiero que sea fácil añadir nuevas interfaces para ampliar la biblioteca en un fúturo.
RNF4	HU14 - Como creador de la biblioteca quiero tener un ejemplo de juego para realizar pruebas sobre él. HU15 - Como creador de la biblioteca quiero tener un modelo que haga de jugador para entrenarlo sobre el juego de ejemplo. HU16 - Como creador de la biblioteca quiero tener un algoritmo que resuelva el juego para usarlo como oponente.
RNF5	HU17 - Como creador de la biblioteca quiero que el ejemplo de juego sea complejo para obtener datos de interés.

Tabla 4.1: Historias de usuario

En la tabla 4.1 se muestran las historias de usuario obtenidas tras analizar los requisitos iniciales, estas historias de usuario forman el *product backlog* del proyecto, así como, una vez repartidas, los distintos *sprint backlogs*.

4.1.2 *Sprint* 1

El primer *sprint* es un caso especial en este desarrollo. Debido al alto contenido teórico de la propuesta, se decide utilizar este *sprint* para analizar cómo llevar la estructura de los juegos de mesa al diseño de la biblioteca de la mejor forma posible, así como al propio diseño y arquitectura de la biblioteca.

Para conseguir esto, se asignan las historias de usuario **HU11**, **HU12** y **HU13**. Posteriormente se analizan y se dividen en tareas que a su vez pueden ser divididas en subtareas si abarcan demasiado.

Tras analizar la historia de usuario **HU11** se obtienen las siguientes tareas:

- Análisis de varios juegos de mesa de ejemplo.
  - Análisis del Risk.

  - Análisis del Catan.
  - Análisis del ajedrez.
- Extracción de los puntos comunes entre distintos juegos de mesa.

Tras analizar la historia de usuario **HU12** se obtienen las siguientes tareas:

- Análisis de la arquitectura de la biblioteca.
- Diseño de la arquitectura de la biblioteca.

Tras analizar la historia de usuario **HU13** se obtienen las siguientes tareas:

- Creación del diagrama de clases.
  - Análisis de las clases.
  - Creación del diagrama en UML.

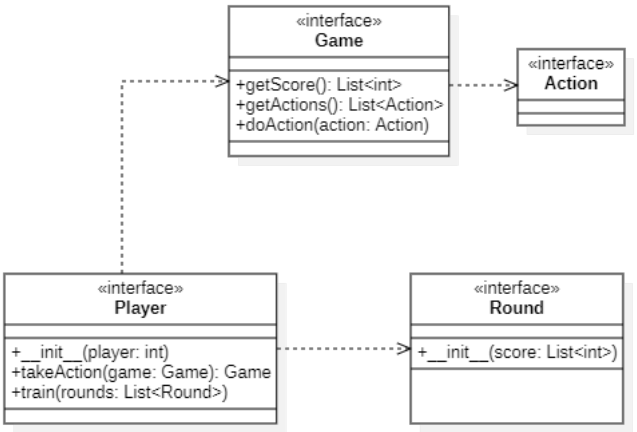

Figura 4.1: Diagrama de clases diseñado durante el primer *sprint*.

La Figura 4.1 muestra el diseño obtenido tras los análisis realizados a lo largo de este *sprint*. En él se pueden observar las tres grandes entidades: "Jugador", "Juego" y "Ronda"; que definen a todo juego de mesa. Además se ha añadido una interfaz "Acción" que se deja a implementar al completo por el desarrollador, ya que cada juego tiene un tipo de acción distinto.

Durante la reunión de revisión del *sprint* se decide que las tareas habían sido sobreestimadas, ya que se realizaron de forma holgada en el tiempo establecido. Por esto, se comenta la posibilidad de aumentar la cantidad de trabajo durante los siguientes *sprints*.

4.1.3 *Sprint* 2

El segundo *sprint* se dedica a la implementación de la biblioteca y su API. Durante la reunión de planificación se decide que, al contrario de como se había planificado inicialmente, se haga todo el desarrollo de la biblioteca durante este *sprint*.

Para conseguir esto se asignan las historias de usuario **HU1**, **HU3** y **HU5**.

Tras analizar la historia de usuario **HU1** se obtienen las siguientes tareas:

- Creación de la interfaz "Jugador".
  - Diseño de la interfaz.
  - Implementación de la interfaz.

Tras analizar la historia de usuario **HU3** se obtienen las siguientes tareas:

- Creación de la interfaz "Juego".
  - Diseño de la interfaz.
  - Implementación de la interfaz.

Tras analizar la historia de usuario **HU5** se obtienen las siguientes tareas:

- Creación de la interfaz "Ronda".
  - Diseño de la interfaz.
  - Implementación de la interfaz.

Como tarea asociada a todas las historias de usuario anteriores se obtiene:

- Creación de la clase "Partida".

- Diseño de la clase.
- Implementación de la clase.
- Implementación del método “jugar”.

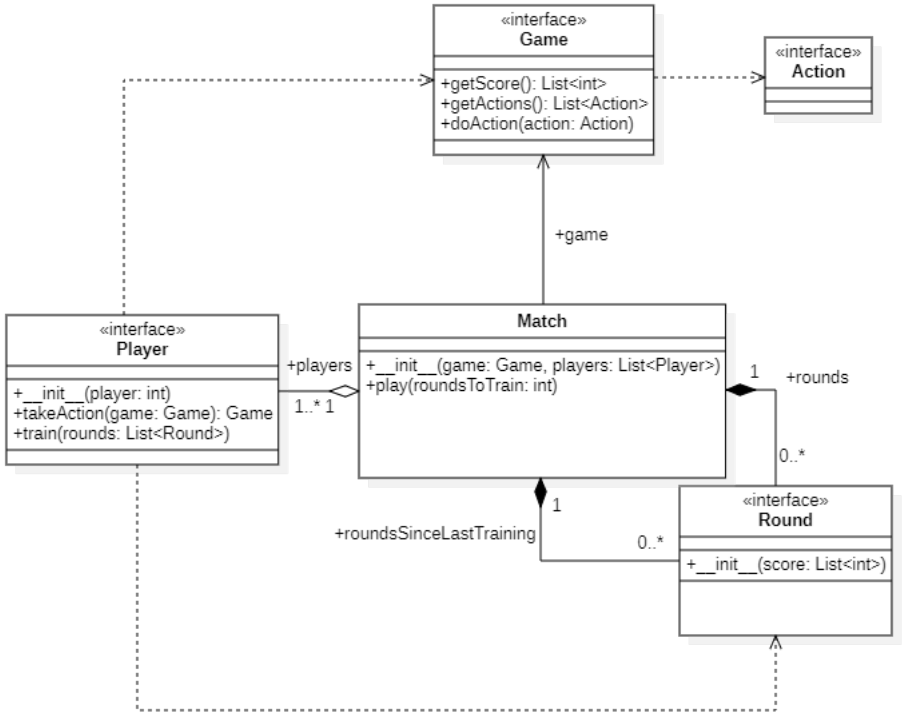

Figura 4.2: Diagrama de clases tras el segundo *sprint*.

Una vez programadas las interfaces principales se ve que es necesario crear la clase “Partida” con el objeto de gestionar, como bien indica su nombre, una partida de un juego con unos jugadores dados. Esa “Partida” está formada por una serie de rondas al juego asociado, a su vez, las rondas son jugadas por el conjunto de jugadores de la “Partida”. Esto se puede observar en la Figura 4.2.

Durante la reunión de revisión del *sprint* se decide dejar la biblioteca en el estado en el que se encuentra tras esta primera iteración y pasar a trabajar en la implementación de un ejemplo durante los siguientes *sprints*. En referencia a lo hablado durante la reunión del primer *sprint* se considera que el trabajo durante este se ha estimado de mejor manera.

4.1.4 *Sprint* 3

El tercer *sprint* se dedica al diseño e implementación de un juego de ejemplo. El objetivo es proveer de un ejemplo funcional sobre el que se pudan probar las capacidades de la librería para así ejemplificar como trabajar los modelos y algoritmos para resolverlo. Con el fin de acortar el desarrollo y enmarcarlo en los límites temporales establecidos, en este punto se decide buscar y adaptar un juego ya existente.

Para conseguir esto se asignan las historias de usuario **HU4**, **HU6**, **HU14** y **HU17**.

Tras analizar la historia de usuario **HU4** se obtienen las siguientes tareas:

- Creación de la clase que implementa la interfaz "Juego".

Tras analizar la historia de usuario **HU6** se obtienen las siguientes tareas:

- Creación de la clase que implementa la interfaz "Ronda".

Tras analizar la historia de usuario **HU14** se obtienen las siguientes tareas:

- Diseño del juego.
  - Diseño del tablero del juego.
  - Diseño de las reglas del juego.
  - Diseño de las unidades del juego.
  - Diseño de las edificios del juego.
  - Diseño de las acciones del juego.
- Implementación del juego.
  - Implementación del tablero del juego.
  - Implementación de las reglas del juego.
  - Implementación de las unidades del juego.
  - Implementación de las edificios del juego.
  - Implementación de las acciones del juego.
  - Creación de una clase generadora de mapas.

- Creación de mapas del juego.

Tras analizar la historia de usuario **HU17** se obtienen las siguientes tareas:

- Análisis de posibles juegos a implementar.

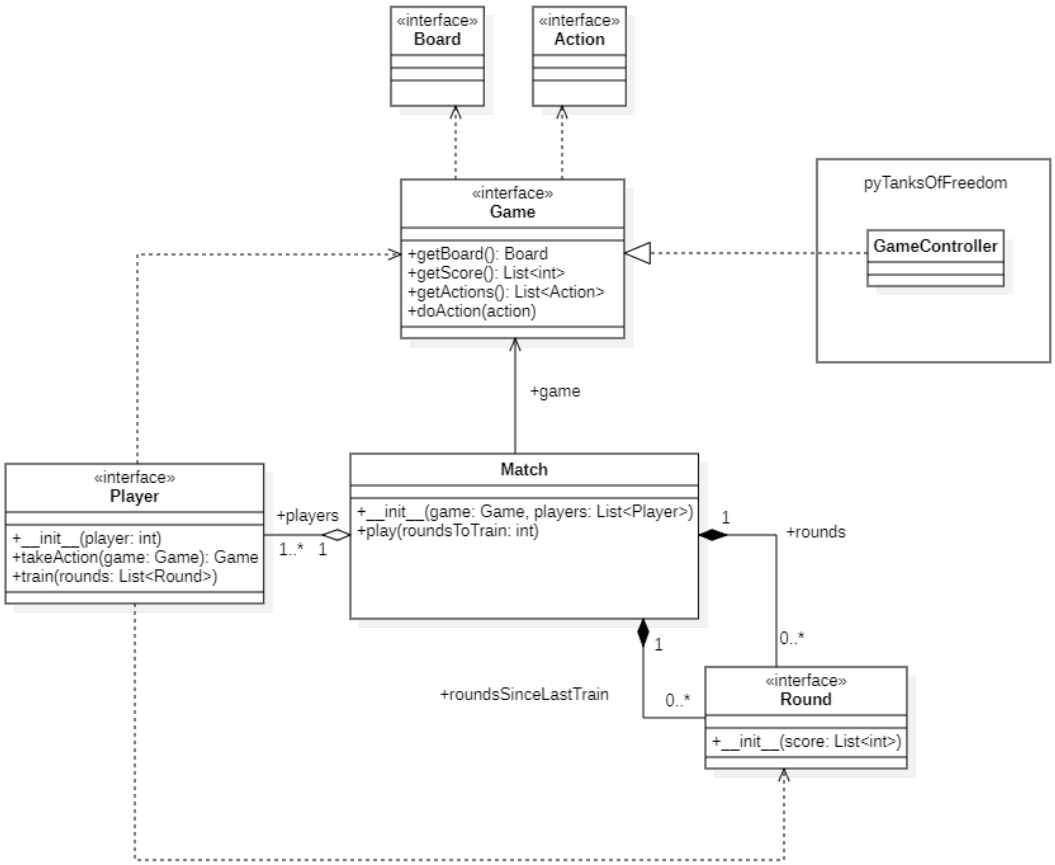

Figura 4.3: Diagrama de clases tras el tercer *sprint*.

Al finalizar el tercer *sprint* se tiene un juego programado del cuál se habla en mayor detalle a lo largo del capítulo 5. En la Figura 4.3 se puede ver como implementa la interfaz "Juego", el diagrama de clases correspondiente al juego se encuentra en el apartado 5.1.

Durante la reunión de revisión del *sprint* se considera que el juego cumple con la complejidad necesaria para la realización de pruebas sobre él. Además, se comprueba que la interfaz

podría incluir también un método para obtener el tablero, ya que es un elemento común entre todos los juegos de mesa que muestra el estado de los mismos.

4.1.5 *Sprint* 4

El cuarto *sprint* se concentra a la implementación de los jugadores para el ejemplo desarrollado en el anterior *sprint*. Implementando tanto un jugador que mediante un algoritmo como otro que utilice un modelo basado en redes neuronales.

Para conseguir esto se asignan las historias de usuario **HU2**, **HU15** y **HU16**.

Tras analizar en conjunto la historia de usuario **HU4** con **HU15** y **HU16** se obtienen las siguientes tareas:

- Creación de la clase que implementa la interfaz "Jugador" para el uso de un *Monte Carlo Tree Search* (a partir de ahora MCTS).
- Creación de la clase que implementa la interfaz "Jugador" para el uso de una red de neuronas.

Tras analizar la historia de usuario **HU15** se obtienen las siguientes tareas:

- Implementación de un MCTS para resolver el juego.

Tras analizar la historia de usuario **HU16** se obtienen las siguientes tareas:

- Implementación de una red de neuronas para resolver el juego.
  - Definir datos de entrada y salida.
  - Definir la estructura del modelo.
  - Implementación de un método que genere el modelo.

También se crea una tarea con el fin de añadir el método de obtención de tablero a la interfaz como se comenta durante la reunión de revisión del *sprint* anterior:

- Añadir tablero a la interfaz "Juego".

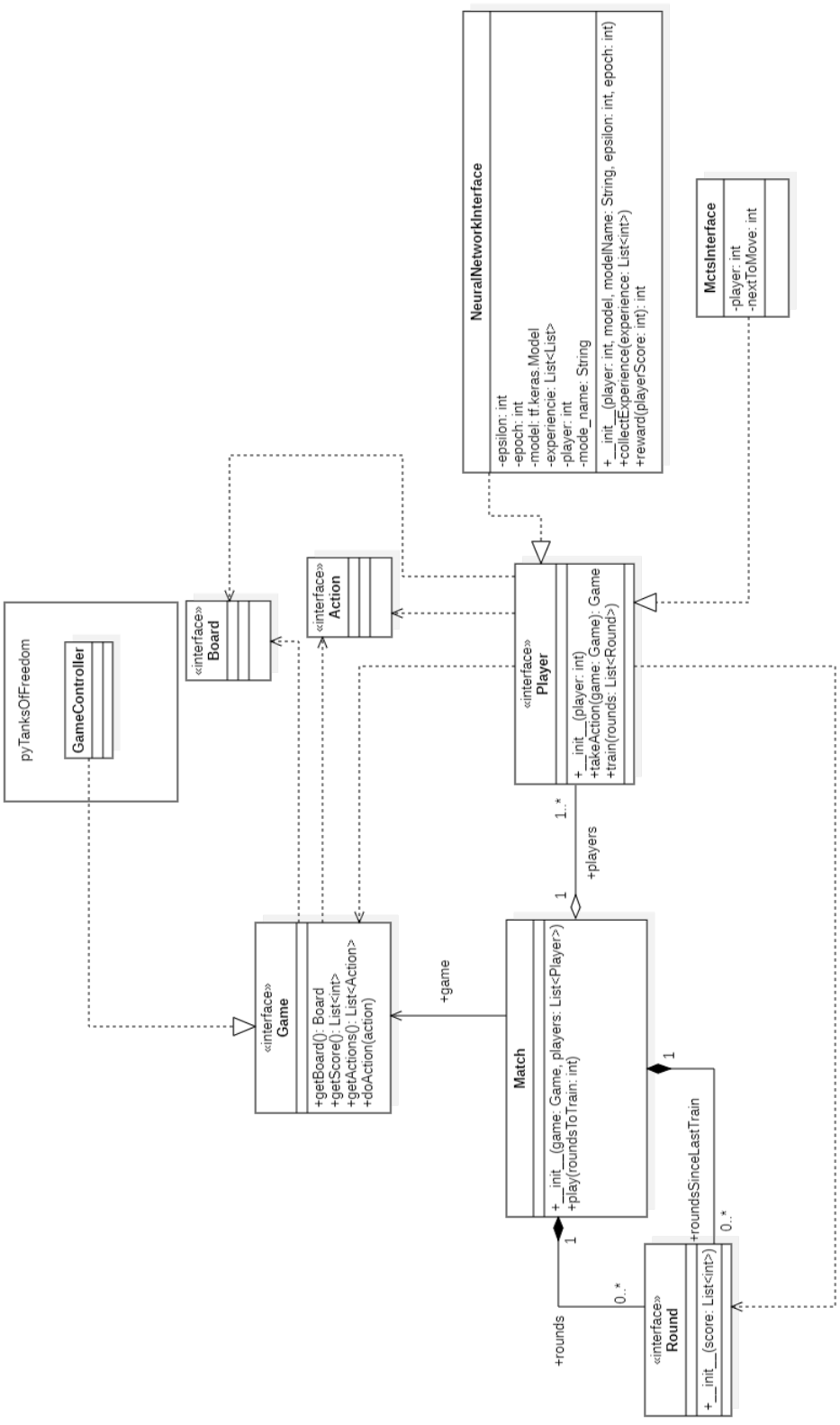

Figura 4.4: Diagrama de clases tras el cuarto *sprint*.

Durante este *sprint* se añade una nueva interfaz llamada "Tablero", similar a la interfaz de "Acción" que también se deja para ser implementada con completa libertad, véase la Figura 4.4. Además, se crean dos clases que implementan la interfaz "Jugador", añadiendo métodos en los casos necesarios. Gracias a esto también se puede comprobar que los jugadores también interactúan con la interfaz "Acción" y la interfaz "Tablero".

Durante la reunión de revisión del *sprint* se comprueba que la interfaz de "Jugador" implementada con anterioridad encaja con los jugadores que se han implementado a lo largo del *sprint*.

4.1.6 *Sprint* 5

El quinto *sprint* se dedica a la implementación de un jugador humano, así como a diversas utilidades para facilitar la posterior realización de pruebas y análisis de los resultados. Se añaden métodos para la generación de informes de datos y para la configuración de los parámetros de las ejecuciones.

Para conseguir esto se asignan las historias de usuario **HU7**, **HU8**, **HU9** y **HU10**.

Tras analizar la historia de usuario **HU7** se obtienen las siguientes tareas:

- Análisis de los posibles datos a registrar.
- Implementación del método de recogida y registro de datos.

Tras analizar la historia de usuario **HU8** se obtienen las siguientes tareas:

- Creación de la clase que implementa la interfaz "Jugador" para el uso de un jugador humano.

Tras analizar la historia de usuario **HU9** se obtienen las siguientes tareas:

- Implementación de la interfaz de usuario.

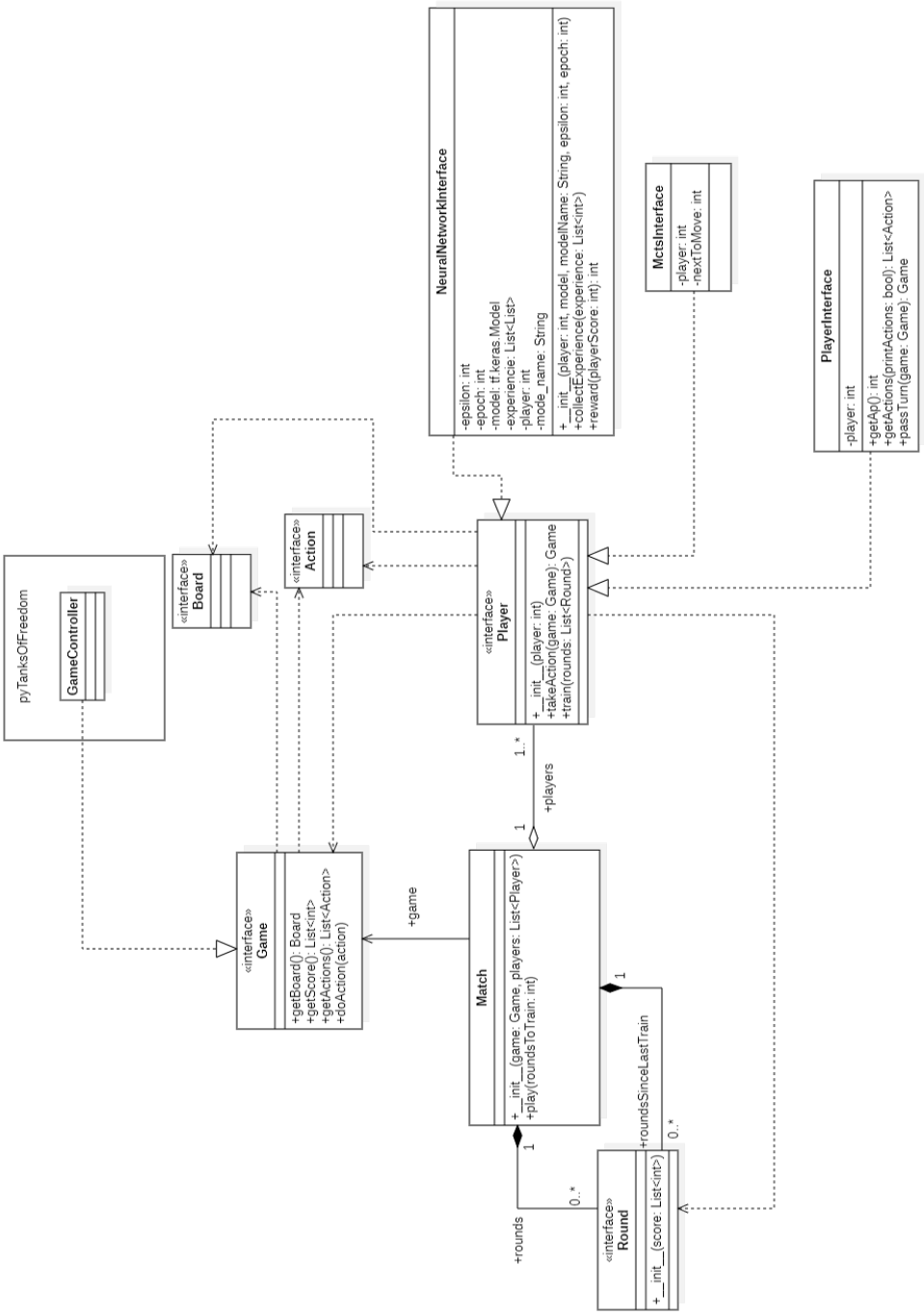

Figura 4.5: Diagrama de clases tras el quinto *sprint*.

Tras analizar la historia de usuario **HU10** se obtienen las siguientes tareas:

- Análisis de los posibles parámetros a configurar.
- Implementación del método de configuración.
- Creación del archivo de configuración.

La Figura 4.5 muestra la nueva clase de "Jugador" desarrollada durante este *sprint*. Dicha clase permite interactuar a un jugador humano con el juego y para ello implementa ciertos métodos que lo facilitan. Por otro lado, los métodos de recogida de datos y de configuración no se reflejan en este diagrama.

Durante la reunión de revisión del *sprint* se considera que la interfaz de usuario es valida, así como que los datos que se obtienen de los enfrentamientos aportan suficiente información y que el archivo de configuración es de utilidad.

4.1.7 *Sprint* 6

Este último *sprint* se concentra en la realización de las pruebas con los ejemplos implementados, así como a la recogida de los datos generados y su análisis. En el siguiente capítulo se comenta en mayor detalle cómo han sido los resultados, además de analizarlos.

En este caso no se han asignado historias de usuario al *sprint*, ya que estas pruebas complementan a todas las tareas ya realizadas anteriormente aunque tengan el suficiente peso como para ser tratadas por separado. Aún así durante las pruebas se han realizado pequeños cambios y ajustes a la configuración y se ha retocado el método que genera los datos sobre los resultados con el fin de obtener los más relevantes.

A lo largo de este *sprint* se realizan las siguientes tareas:

- Pruebas de enfrentamiento entre MCTS y modelo.
  - Entrenamiento del modelo
  - Validación del modelo
- Validación del modelo contra un jugador humano.
- Análisis de los resultados.

Las pruebas entre el MCTS y el modelo se realizan de forma iterativa haciendo cambios en la configuración entre cada iteración.

Durante la reunión de revisión del *sprint* se analizan y discuten los resultados obtenidos, así mismo también se plantean las posibles líneas de desarrollo que se puedan abrir a continuación de este trabajo.

Capítulo 5

Ejemplo de experimentación

Para comprobar la utilidad de la biblioteca es necesario realizar un experimento y analizar sus resultados. Para su realización se implementa un juego de estrategia en Python, el cual se detalla en el apartado 5.1. La elección del género de estrategia viene propiciada por cómo encaja en los requisitos para el uso de la biblioteca y la facilidad para recoger los datos tras una toma de decisión y, en general, los datos asociados al resultado de cada partida. Este juego sirve de ejemplo de implementación de las interfaces de la biblioteca y permitirá entrenar un modelo de red de neuronas que aprenda a jugar al mismo.

En concreto, el juego seleccionado tiene por objetivo la conquista de la base rival. Para ello, el juego cuenta con una serie de unidades que pueden moverse, atacar y conquistar edificios tanto rivales como neutrales. Estos edificios son los que generaran en un esquema clásico de tanto los puntos como las unidades.

Para agilizar la ejecución de los enfrentamientos se hacen en el Centro de Super Computación de Galicia (a partir de ahora CESGA). Su gran capacidad de computo permite la ejecución de cientos de enfrentamientos de una forma más rápida que en un ordenador personal.

Se registran todos los resultados de los enfrentamientos realizados para su posterior análisis. El objetivo es comprobar cómo de bien funciona el modelo tras ser entrenado con este método. Por esta razón se escogen como oponentes al algoritmo MCTS, que es uno de los más usados para las inteligencias artificiales en juegos de estrategia, y a un jugador humano con ciertos conocimientos sobre el juego.

5.1 Adaptación de un juego de estrategia

Con el fin de agilizar el proceso se decide buscar un proyecto de código abierto para realizarejemplificar el uso de la biblioteca. Tras analizar varias posibilidades, se decide utilizar Tanks Of Freedom[18]. Se escoge ya que es un juego equilibrado, donde ambos jugadores parten del mismo punto inicial, pero también permite, según el mapa, darle mayor ventaja o desventaja a alguno de los jugadores. Esto ayuda a ver cómo reacciona cada jugador a situaciones completamente distintas.

En este juego de estrategia por turnos, los jugadores deben conquistar la base del otro jugador para ganar la partida. El juego transcurre en un campo de batalla formado por una rejilla, en cada casilla puede haber un obstáculo, un edificio, una unidad o estar vacío. Los edificios generan puntos que posteriormente se pueden gastar en crear nuevas unidades en un punto cercano a un edificio encargado de reclutarlas. Además, pueden ser neutrales o pertenecer a alguno de los jugadores, bien porque comenzó bajo su control o porque fue conquistado. Las unidades pueden moverse por el campo de batalla, atacar a otras unidades o conquistar edificios; perteneciendo siempre a alguno de los jugadores. Un ejemplo de un campo de batalla durante una partida puede verse en la Figura 5.1.

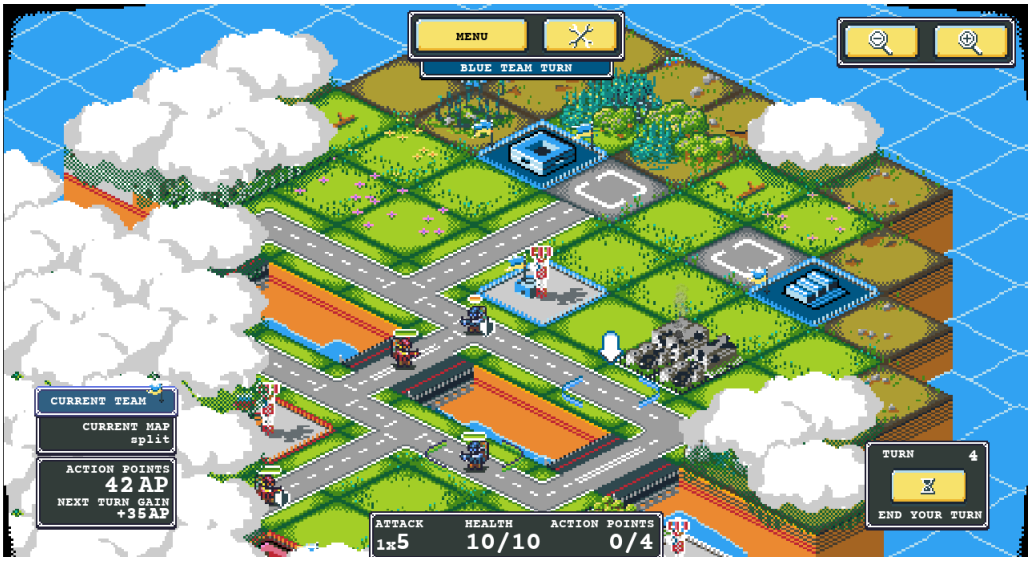

Figura 5.1: Campo de batalla de Tanks Of Freedom.

Los edificios del juego pueden ser de los siguientes tipos:

- Cuartel general. Es la base del jugador, si es conquista por el oponente perderá el juego.

Permite crear soldados por un coste de 40 puntos y genera 20 puntos por turno. Al capturarse se destruye la unidad que lo captura.

- Barracones. Permite crear soldados por un coste de 25 puntos. Al capturarse se destruye la unidad que lo captura.
- Taller. Permite crear tanques por un coste de 50 puntos. Al capturarse se destruye la unidad que lo captura.
- Aeropuerto. Permite crear helicópteros por un coste de 70 puntos. Al capturarse se destruye la unidad que lo captura.
- Torre. Genera 15 puntos por turno.

Por otro lado tenemos las unidades, que pueden ser de los siguientes tipos:

- Soldado. Puede moverse hasta 4 casillas por turno, realiza 5 puntos de daño por ataque y tiene 10 puntos de vida. Es la única unidad que puede capturar edificios.
- Tanque. Puede moverse hasta 6 casillas por turno, realiza 10 puntos de daño por ataque y tiene 15 puntos de vida.
- Helicóptero. Puede moverse hasta 8 casillas por turno, realiza 8 puntos de daño por ataque y tiene 10 puntos de vida.

Por último, las acciones que se pueden realizar durante un turno son las siguientes:

- Mover una unidad. Al seleccionar una unidad que no se haya movido aún durante este turno su número máximo de casillas es posible seleccionar una casilla destino dentro del rango de movimiento de la misma.
- Atacar con una unidad. Al seleccionar una unidad que aún pueda desplazarse una casilla más y que se encuentre adyacente a la unidad de un oponente es posible seleccionar la casilla en la que se encuentra dicha unidad y atacarla.
- Capturar un edificio. Al seleccionar una unidad que aún pueda desplazarse una casilla más y que se encuentre adyacente a un edificio que no pertenece al jugador es posible seleccionar la casilla en la que se encuentra dicho edificio y capturarlo.
- Crear una unidad. Al seleccionar un edificio y si se tienen puntos suficientes se podrá crear una nueva unidad del tipo que permita el edificio.

También es posible finalizar el turno en cualquier momento, pero esto no se considera una acción como tal.

Tanks of Freedom está desarrollado utilizando Godot, un motor de videojuegos gratuito y de código abierto que permite el desarrollo de juegos 2D y 3D, pero para el uso del juego con la biblioteca se decide implementar el mismo directamente en Python para agilizar la realización de las pruebas. También se toma la decisión de no realizar la interfaz gráfica del juego, ya que se considera que no es relevante para la realización de las pruebas. Por lo tanto, se implementan tan solo las reglas del juego haciendo las adaptaciones necesarias para desacoplar la lógica de la interfaz de usuario.

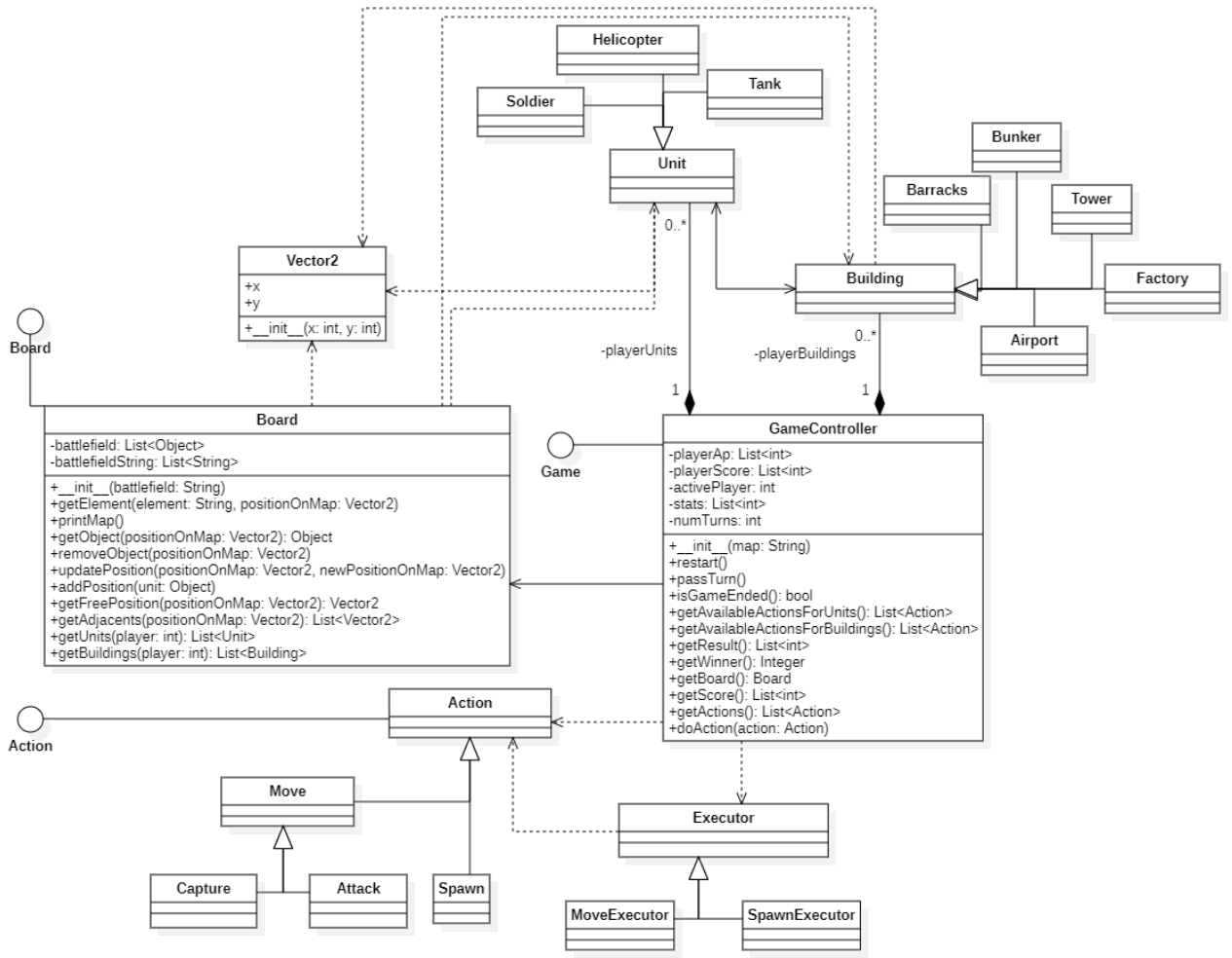

Figura 5.2: Diagrama de clases de control del juego de ejemplo implementado.

En la Figura 5.2 se puede ver cómo se ha desarrollado una clase encargada del control del juego. Dicha clase implementa la interfaz "Juego" de la biblioteca. Realiza 4 grandes funcionalidades: controla las estadísticas de la partida, genera las acciones posibles en un momento dado teniendo en cuenta las reglas del juego y el estado del tablero, ejecuta una acción actualizando el tablero y decide el ganador de la partida. También se puede ver la clase tablero, que se encarga de gestionar el campo de batalla, controlando los edificios y unidades que hay en el mismo,

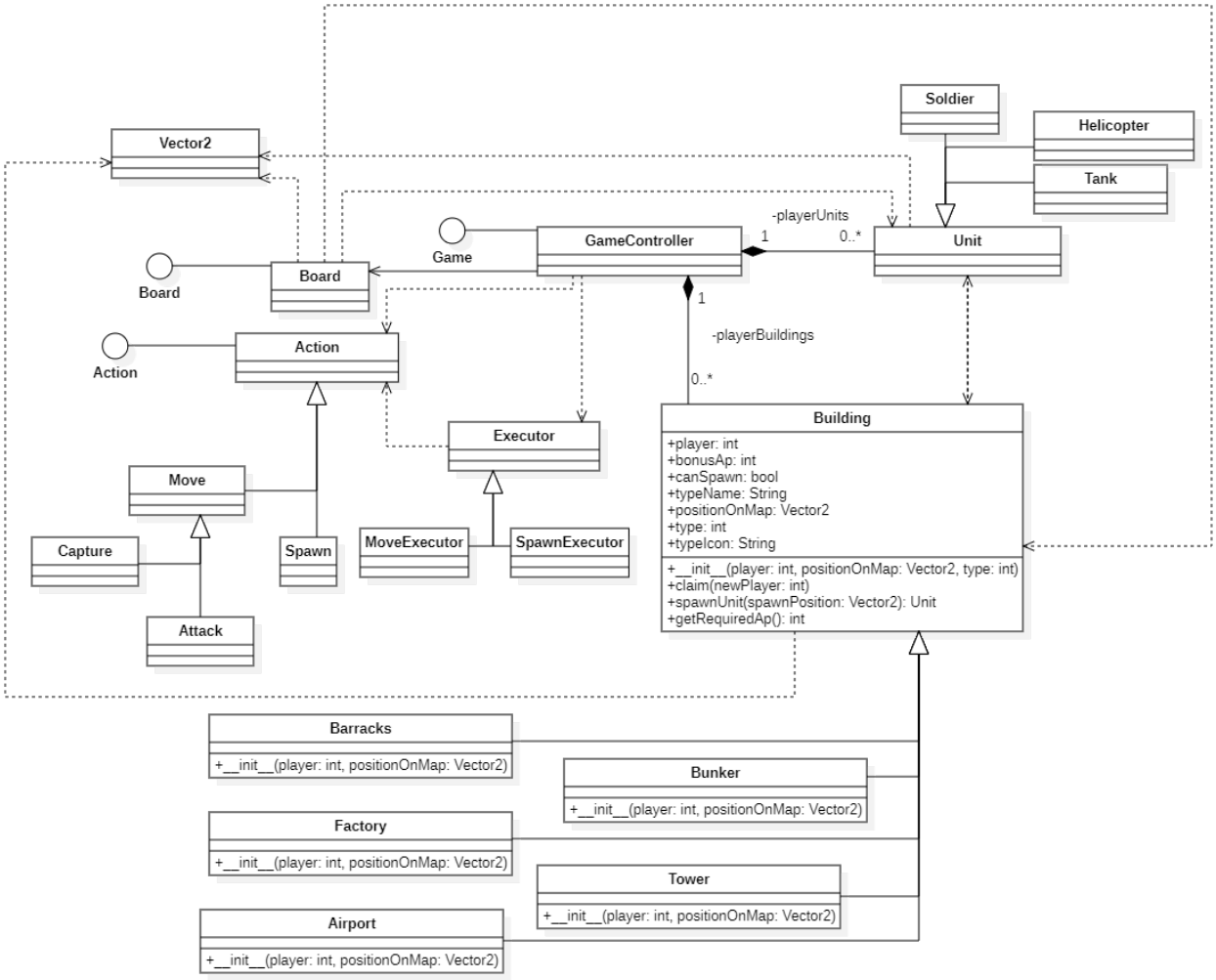

Figura 5.3: Diagrama de clase "Edificio" del juego de ejemplo implementado.

Se ha desarrollado una clase abstracta "Edificio" (*Building*) y de ella heredan las clases de "Búnker" (equivalente a Cuartel general en el juego), "Barracones" (*Barracks*), "Taller" (*Factory*), "Aeropuerto"(*Airport*) y "Torre"(*Tower*); como se puede ver en la Figura 5.3.

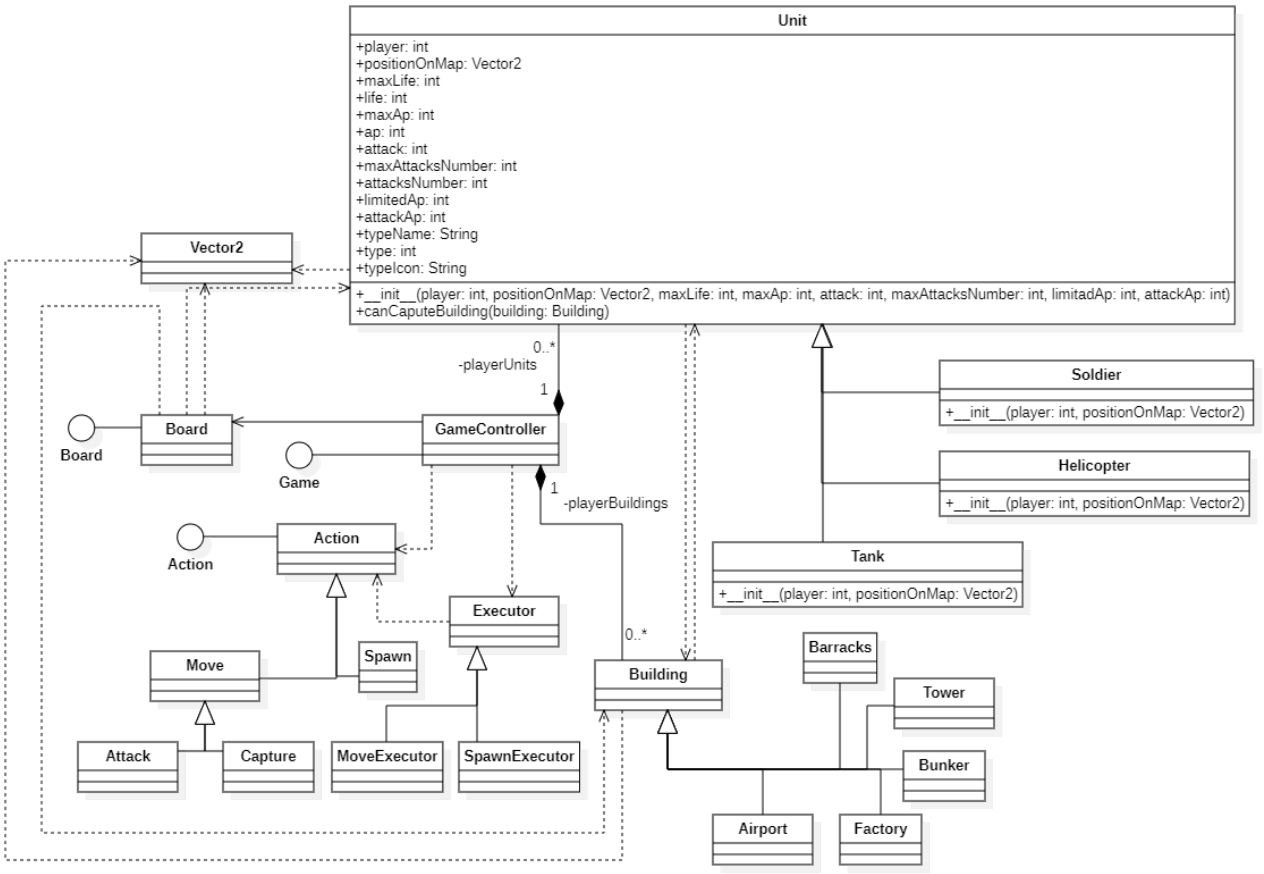

Figura 5.4: Diagrama de clase “Unidad” del juego de ejemplo implementado.

Para las unidades también se ha implementado una clase abstracta “Unidad”(*Unit*). De esta heredan las clases “Soldado”(*Soldier*), “Tanque”(*Tank*) y “Helicópter”(*Helicopter*); como se observa en la Figura 5.4.

Tanto las clases que heredan de “Unidad” como de “Edificio” son clases con muy poca lógica, más centradas en contener las características de los objetos. La lógica del juego se reparte tanto en la clase “Juego”, encargada de la gestión de las reglas”, como en “Tablero” que controla todas las unidades y edificios que están en el campo de batalla.

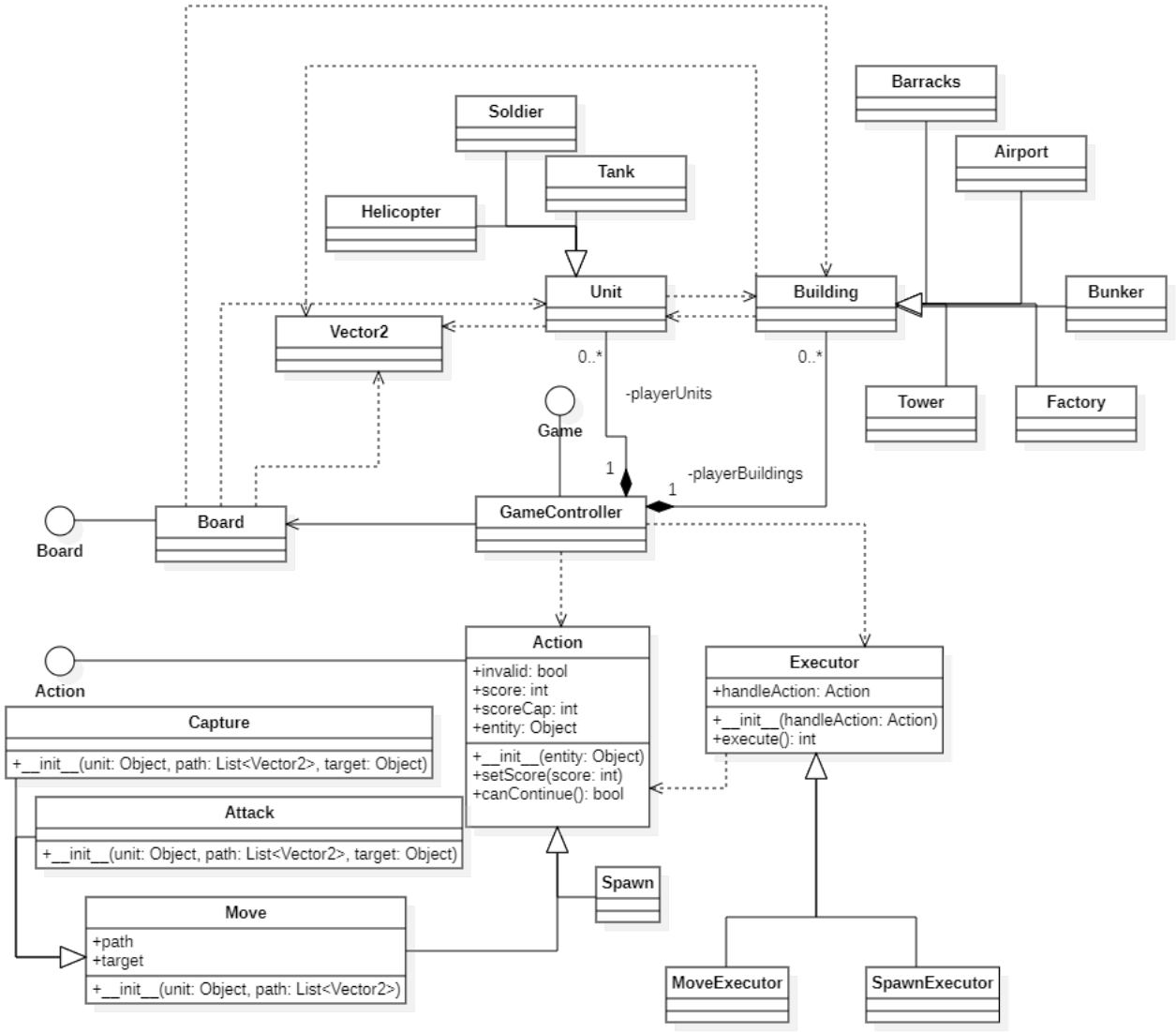

Figura 5.5: Diagrama de clase “Acción” y “Ejecutor” del juego de ejemplo implementado.

En la Figura 5.5 se puede ver que las acciones tienen una clase padre común “Acción”(*Action*), de ella heredan “Mover”(*Move*) y “Crear”(*Spawn*). De “Mover” heredan, a su vez, las clases “Capturar”(*Capture*) y “Atacar”(*Attack*). Además, también se pueden ver las clases que heredan de “Ejecutor”(*Executor*), que son las encargadas de ejecutar una acción cuando un jugador decide realizarla. Hay una clase ejecutor encarga de las acciones de movimiento y otro para las acciones de creación de unidades, ya que son procesos completamente distintos.

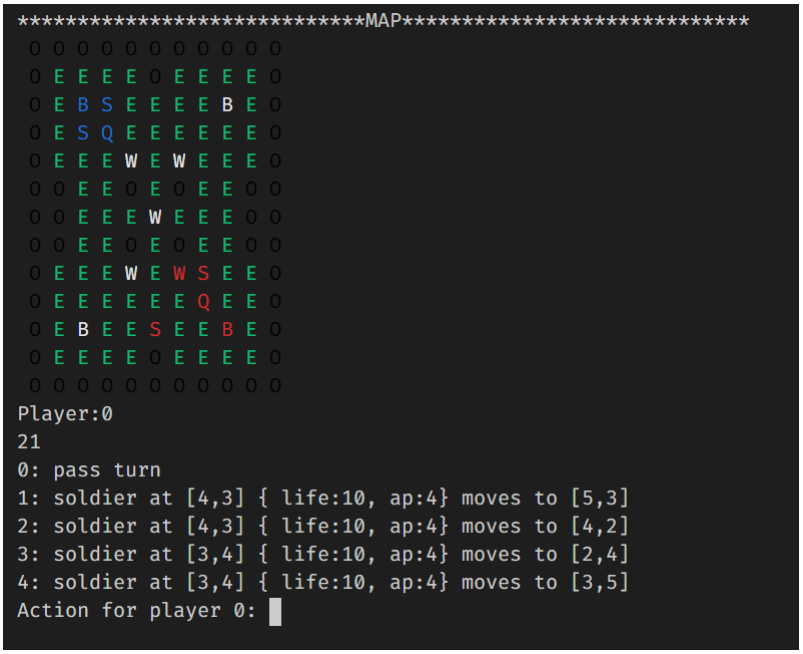

Figura 5.6: Interfaz para el jugador por terminal.

Una vez implementadas todas estas clases solo es necesario desarrollar las clases encargadas de jugar, es decir los jugadores. Para los jugadores se ha programado una interfaz mucho más sencilla que la del juego original, como se muestra en la Figura 5.6.

5.2 Configuración en el CESGA

Con el fin de agilizar la ejecución de los entrenamientos y las validaciones del modelo de red neuronal se ha solicitado acceso al CESGA. Al tratarse de un proyecto con una duración tan limitada, donde solo es posible dedicarle un *sprint* a todo el proceso necesario para realizar las pruebas, se requiere paralelizar los procesos y ejecutarlos a la mayor velocidad posible. El CESGA permite ejecutar en paralelo un gran número de partidas, reduciendo en gran medida el tiempo necesario para entrenar un modelo.

Este problema con el tiempo de ejecución viene causado, sobretodo, por el algoritmo MCTS. Los turnos pueden llegar a durar 30 minutos cuando la lista de posibles acciones alcanza un elevado tamaño. Esto provoca que haya partidas que duren unas 10 horas. Ejecutar una partida de 10 horas en local con solo dos semanas de duración de *sprint* provocaría que los resultados con el modelo que está siendo entrenado estuviesen muy alejados de lo deseado.

Por otro lado, la ejecución de enfrentamientos entre dos modelos de red neuronal en local consume toda la RAM disponible, por lo que termina por bloquear el proceso. Mientras que en el CESGA la RAM es más que suficiente para este tipo de ejecuciones.

Para la ejecución es necesario programar un pequeño *script* que se encarga de lanzarla para cada partida. En él se configura el nombre de los trabajos de ejecución que se generarán por cada una. También se configura el número de ejecuciones a realizar, así como el tiempo que se espera que tarde cada una de ellas. Por otro lado, se configuran los nodos en los que se desea que se ejecute y la memoria que se necesita para ello. Toda esta configuración se muestra en la Figura 5.7, además de cómo se importan los paquetes necesarios por dependencias del programa principal.

```
1  #!/bin/bash
2
3  #SBATCH --job-name=n2
4  #SBATCH --array=1-100
5  #SBATCH --time=10:00:00
6  #SBATCH -p shared
7  #SBATCH --qos shared
8  #SBATCH --mem 16GB
9
10 module load gcccore/6.4.0 python/3.7.7
11 module load numpy/1.18.1-python-3.7.7
12 module load tensorflow/2.0.1-python-3.7.7
13
14
15 python main.py $SLURM_ARRAY_TASK_ID
16
```

Figura 5.7: Código para la ejecución en el CESGA.

Se han lanzado miles de ejecuciones con el fin de conseguir un buen entrenamiento y una buena validación. Esta cantidad de ejecuciones sin el uso del CESGA habrían tardado muchas más semanas en finalizar. Por esto, para este proyecto el uso de la capacidad computacional que ofrece el CESGA es de gran importancia y utilidad.

5.3 Pruebas realizadas

En un proyecto de este estilo las pruebas alcanzan una importancia a la altura del propio desarrollo de la biblioteca. Se requiere comprobar la utilidad y efectividad de la misma con un caso de ejemplo, para comprobar si todas las teorías realizadas durante el trabajo son más o menos correctas. Además, sirve también como ejemplo para todo aquel desarrollador que desee utilizar la biblioteca creada.

Para las pruebas se da uso al juego implementado realizando diversos enfrentamientos entre un modelo y otro adversario. El objetivo es comprobar cómo aprende el modelo a lo largo de los enfrentamientos y cuál es su rendimiento.

Con el fin de tener una mayor variedad de datos para analizar se decide realizar tres tipos de pruebas:

1. Enfrentamientos contra el algoritmo MCTS
2. Enfrentamientos contra otro modelo de IA
3. Enfrentamientos contra un jugador humano

La primera parte de las pruebas han enfrentado al modelo a desarrollar contra el algoritmo MCTS, con esto se busca que el modelo aprenda de las decisiones que toma el algoritmo. Se han realizan miles de enfrentamientos cambiando el mapa en el que se enfrentan para evitar el sobre entrenamiento. El cambio de mapa se realiza de dos formas, secuencial o aleatoria. El total de partidas jugadas entre el modelo y el MCTS ha sido de 2924. En un principio se realizaron 300 partidas con selección de mapa secuencial y 300 aleatoria, tras esto se comparó los resultados y se decidió que el resto de partidas se ejecutarían con selección aleatoria.

Tras eso el mejor modelo resultante de la fase anterior se ha enfrentado a un modelo nuevo. Con esto se intenta comprobar si el primer modelo ha sido o no sobre entrenado. En este caso se ejecutan unos cientos de enfrentamientos, manteniendo el cambio de mapa como durante los enfrentamientos anteriores. El total de las partidas asciende a 210 de las cuales en 108 se ha cambiando de forma aleatoria de mapa y 102 de forma secuencial.

El último paso durante las pruebas es el enfrentamiento contra un jugador humano, se han realizado 10 partidas para verificar la efectividad del modelo en este juego. Para esto se escoge el mapa con mejores resultados y los 10 enfrentamientos se efectúan en él.

A modo de resumen, en la tabla 5.1 se muestra el número de partidas realizadas contra cada oponente y divididas según el tipo de selección de mapa.

	Secuencial	Aleatoria
MCTS	363	2561
Modelo nuevo	102	108
Jugador humano	10	0

Tabla 5.1: Partidas por oponente y tipo de selección de mapa

5.4 Discusión de resultados

De cada partida ejecutada durante las pruebas se han guardado los siguientes datos:

- Puntuación de los jugadores
- Tipo de jugadores
- Mapa
- Orden de selección de los mapas
- Duración en turnos
- Jugador ganador

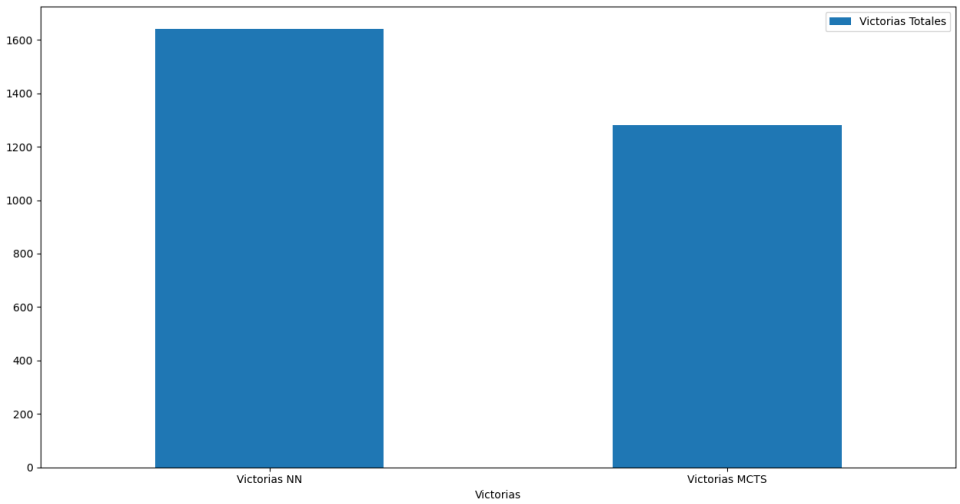

Figura 5.8: Número de victorias en los enfrentamientos contra MCTS.

Tras realizar todos los enfrentamientos contra el algoritmo MCTS se comprueba el número de veces que ha ganado cada uno, que se puede ver en la Figura 5.8. Como primer vistazo sirve para decir que el modelo ha conseguido buenos resultados, 1642 victorias del modelo contra 1282 del MCTS. A esto hay que sumarle que el algoritmo MCTS tiene un tiempo de ejecución muy superior al del modelo.

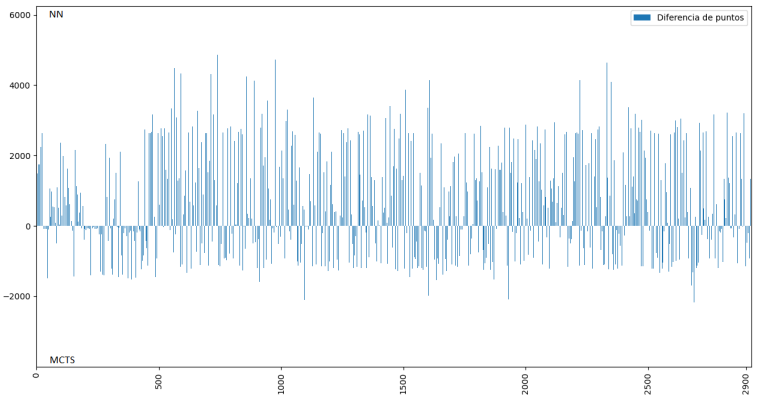

Figura 5.9: Diferencia de puntuación contra MCTS.

Aunque las victorias son un dato bastante importante, es interesante saber si las victorias son por muchos puntos o por pocos. Esto se puede comprobar en la Figura 5.9, en ella se ve la diferencia de puntos en cada partida. El modelo tiende a ganar por una mayor cantidad de puntos que el algoritmo MCTS, que parece que siempre consigue ganar por una cantidad parecida de puntos.

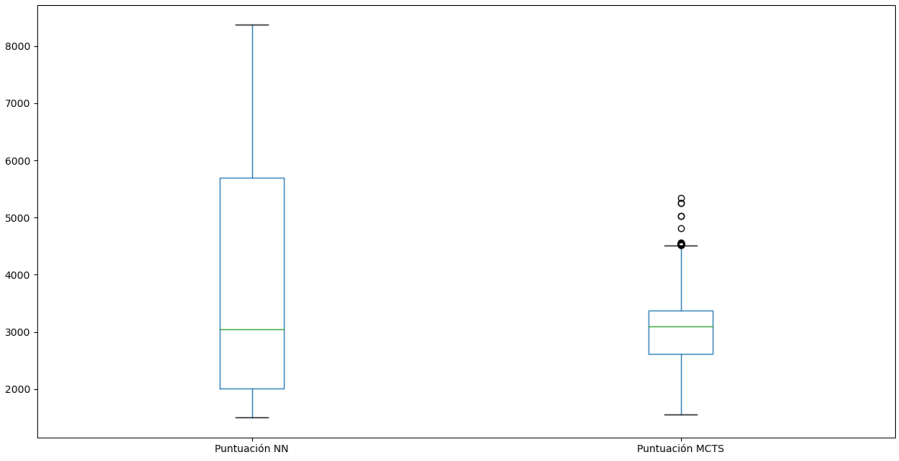

Figura 5.10: Distribución de la puntuación contra MCTS.

En la Figura 5.10 se puede observar que las puntuaciones que consigue el modelo varían mucho más que las del algoritmo, que consigue unas puntuaciones más similares entre parti-

das. Esto explica que siempre gane por una cantidad parecida de puntos.

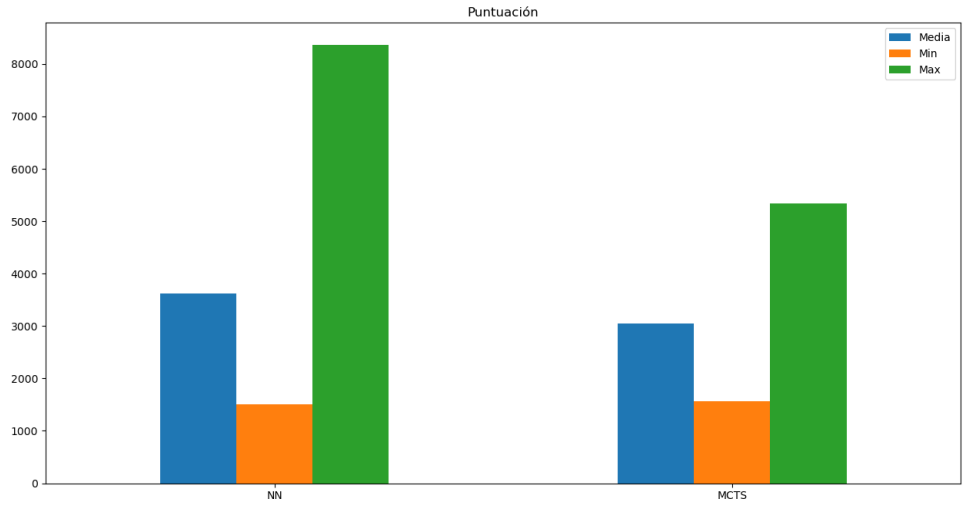

Figura 5.11: Puntuación máxima, mínima y media contra MCTS.

La puntuación del algoritmo MCTS se mantiene más estable, pero la máxima puntuación a lo largo de las partidas la consiguió el modelo de red neuronal con 8380 puntos y, en general, sus puntuaciones son más elevadas consiguiendo una puntuación media superior. Esto se puede observar en la Figura 5.11. Lo comentado sobre la estabilidad del algoritmo MCTS se demuestra en esta figura ya que la diferencia entre el mínimo y el máximo de puntuación es tan solo de 3780 puntos, frente a los 6870 puntos en el caso del modelo.

Una vez se han analizado las partidas de forma general, se debe comprobar que tipo de selección de mapas le da mejor resultado al modelo, para ello debemos comprobar las victorias conseguidas según el tipo.

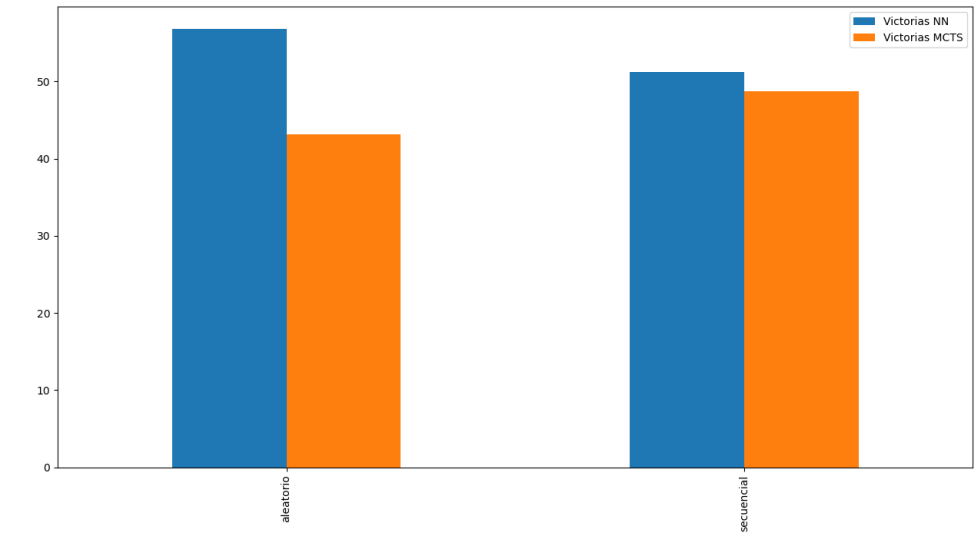

Figura 5.12: Porcentaje de victorias según tipo de selección de mapa contra MCTS.

En la Figura 5.12 se puede observar que el porcentaje de victorias a favor del modelo varía si la selección de mapa es aleatoria o secuencial, dando mejor resultado el primer caso. Esto puede darse por la mayor capacidad de generalización que tiene el modelo al cambiar constantemente de entorno.

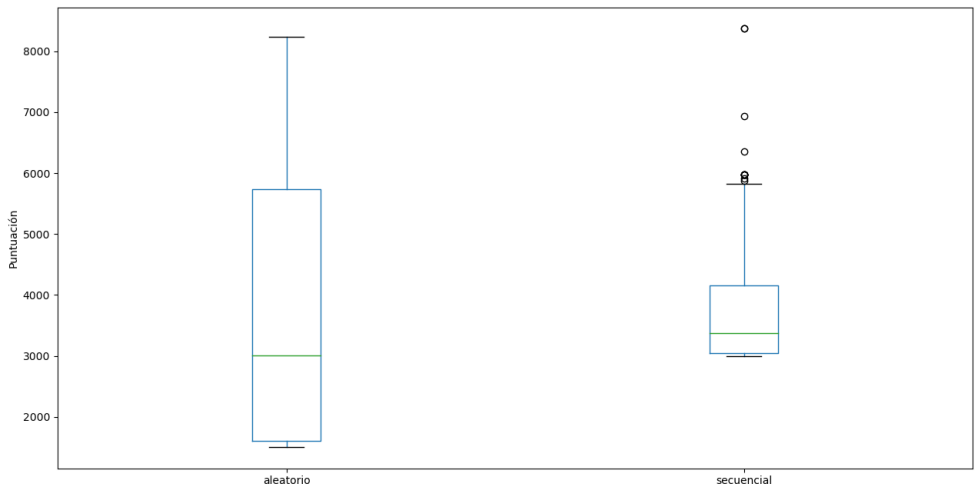

Figura 5.13: Puntuación según tipo de selección de mapa contra MCTS.

Como se ha comentado anteriormente, no solo es interesante conocer la victorias, también lo es saber las puntuaciones obtenidas. Para esto se puede ver en la Figura 5.13 como varía la puntuación según el tipo de selección de mapa con el que se realiza la partida. En las partidas donde el mapa se selecciona de forma aleatoria la puntuación fluctúa más pero tiende a conseguir también puntuaciones más altas. Sin embargo, en las partidas con selección secuencial tiene una media y un mínimo de puntuación mayor, esto puede deberse a un sobre entrenamiento provocado por la repetición constante del mismo mapa.

Tras los enfrentamientos contra el algoritmo MCTS se realiza una segunda fase contra un nuevo modelo para comprobar la eficacia del modelo entrenado contra otro rival. Al mismo tiempo que se genera un nuevo modelo que aprenda del ya existente, con el fin de conseguir un modelo que generalice de la mejor manera posible.

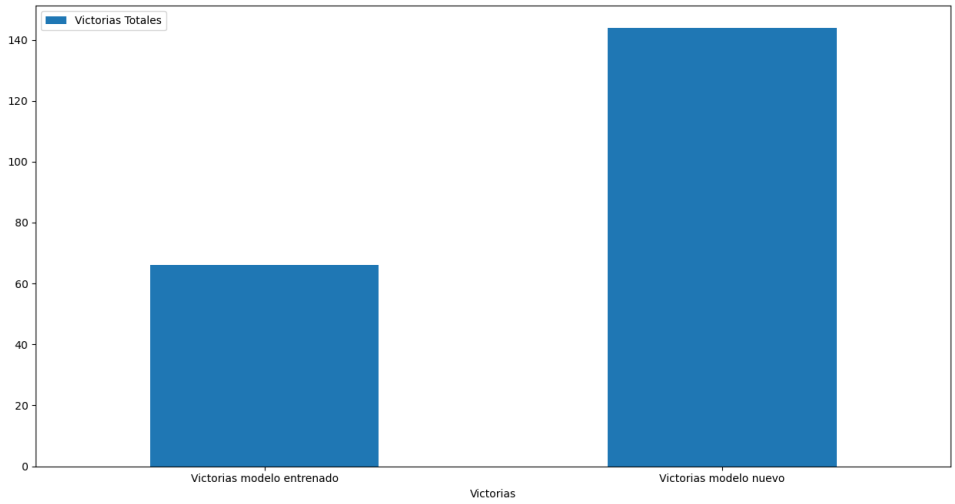

Figura 5.14: Número de victorias en los enfrentamientos contra un nuevo modelo.

Sorprendentemente el nuevo modelo ha conseguido vencer más veces que el entrenado en la anterior fase de pruebas. De las 210 partidas el modelo nuevo se ha afianzado un total de 144. Lo que confirma que el modelo entrenado fue sobre entrenado al enfrentarlo demasiadas veces contra el algoritmo MCTS.

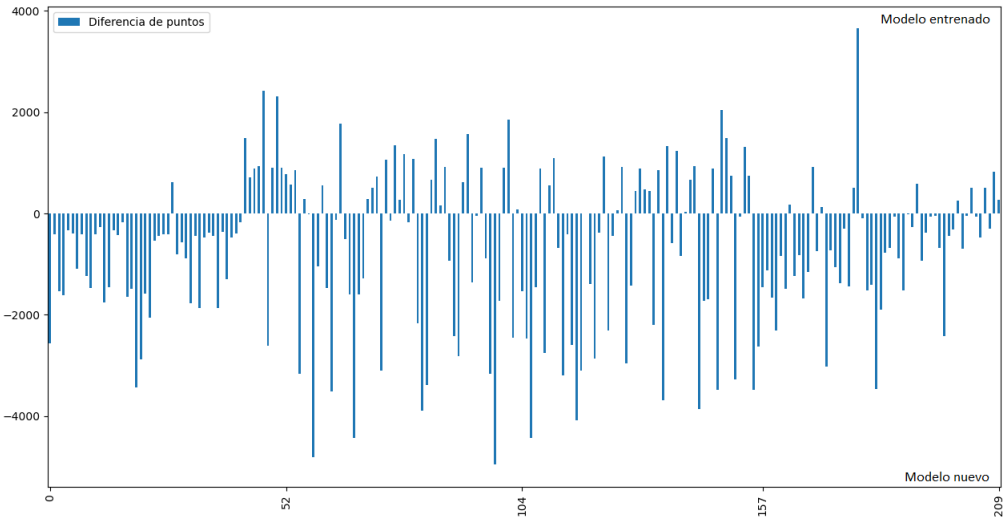

Figura 5.15: Diferencia de puntuación a lo largo del conjunto de partidas contra un nuevo modelo.

A pesar de ser derrotado un número considerable de veces, en la Figura 5.16 se puede observar que el modelo entrenado se va adaptando a su nuevo rival y que al final del conjunto de partidas consigue reducir el número de puntos que obtiene su oponente mientras que aumenta los suyos. Por lo que esta forma de entrenamiento podría facilitar a un modelo corregir este tipo de problemas de sobre entrenamiento.

También destaca como el nuevo modelo gana prácticamente todos los primero enfrentamientos, mientras el antiguo modelo no se adapta a su nuevo rival. En las primeras 50 partidas el nuevo modelo consigue el récord de puntos en una partida conseguidos durante las pruebas con 9560.

Aunque la puntuación máxima sea tan elevada, la media es inferior a los 4000 puntos en ambos modelos y, en el modelo entrenado con anterioridad, la puntuación es más estable, como se puede ver en la Figura 5.16. Aunque, comparado con el MCTS, ningún modelo parece mantener estable el número de puntos por partida, consiguiendo unos extremos de máximo y mínimo entre pocas partidas.

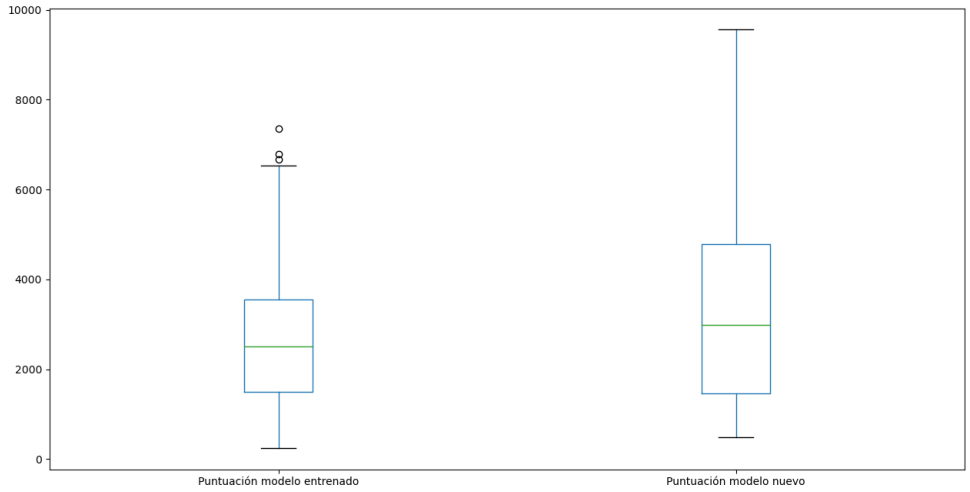

Figura 5.16: Distribución de la puntuación contra un nuevo modelo.

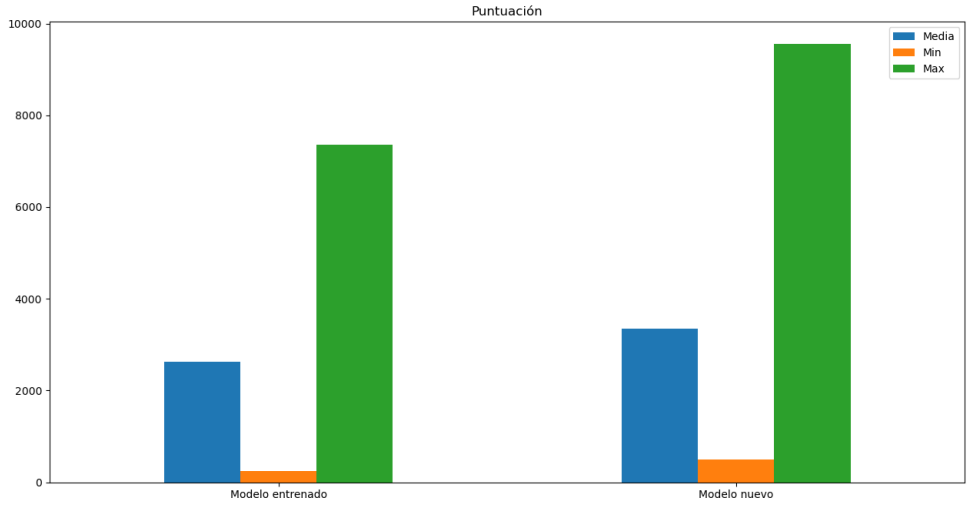

Figura 5.17: Puntuación máxima, mínima y media contra un nuevo modelo.

Los extremos en la puntuación se ven con mayor claridad en la Figura 5.17. Aquí se puede ver como la puntuación mínima de ambos modelos no llega ni a los 1000 puntos y la máxima en ambos casos supera los 7000. Estos mínimos tan bajos disminuyen la puntuación media en comparación con los enfrentamientos contra el MCTS.

Estos mínimos, en algunos casos, ocurren porque uno de los dos jugadores conquista en un turno temprano la base del oponente, finalizando la partida antes de permitir una acumulación sustancial de puntos por parte de ninguno de los jugadores.

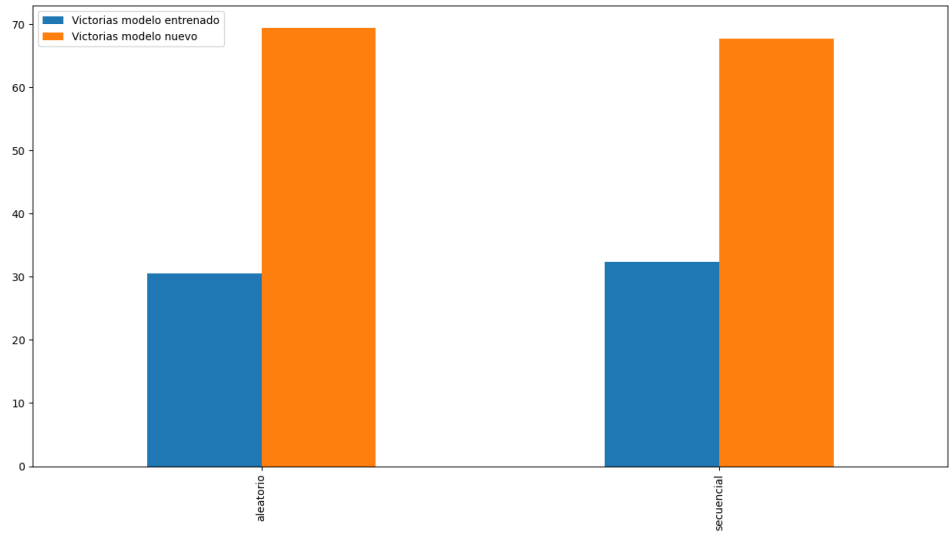

Figura 5.18: Porcentaje de victorias según tipo de selección de mapa contra un nuevo modelo.

En la Figura 5.18 también se puede ver una variación en los porcentajes de victoria según el tipo de selección de mapa. Aunque en esta ocasión es menor y podría estar provocada por la adaptación progresiva del modelo entrenado, ya que las partidas con selección secuencial se realizaron después de las de selección aleatoria.

De nuevo, una mejor forma de ver si realmente el tipo de selección de mapa influye es comparando la puntuación conseguida con cada uno. Si se observa la Figura 5.19 se ve que las mejores puntuaciones se centran en la selección aleatoria, más concretamente en los mapas “four_isles_reverse” y “four_isles”

Por último, durante las pruebas se realizó una serie de 10 enfrentamientos entre un jugador y el modelo entrenado contra el MCTS. Para estos enfrentamientos se escogió el mapa con mejor puntuación buscando un mayor reto para el jugador.

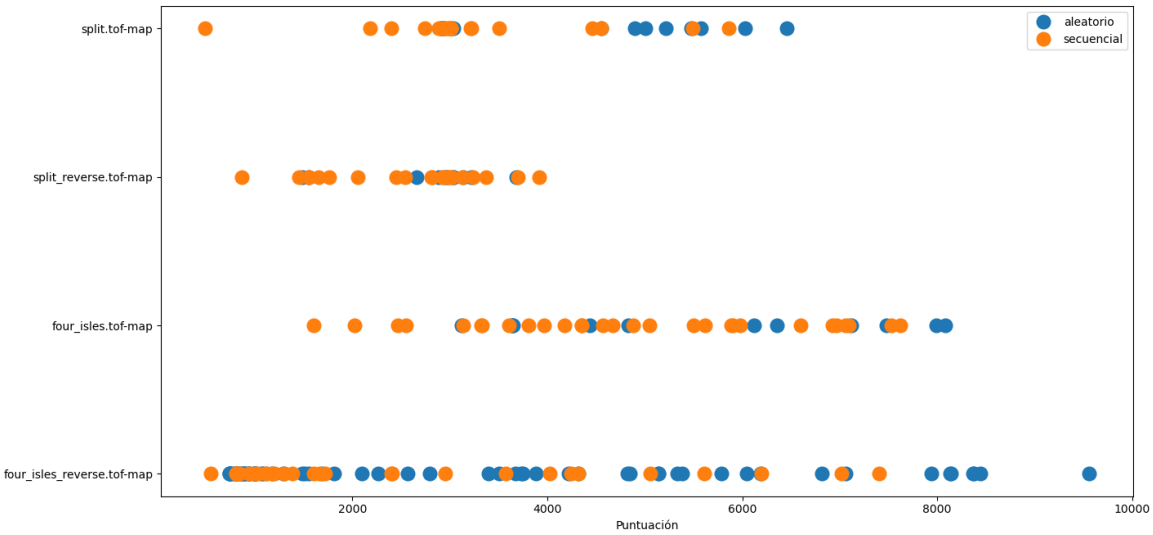

Figura 5.19: Puntuación según tipo de selección y mapa contra MCTS.

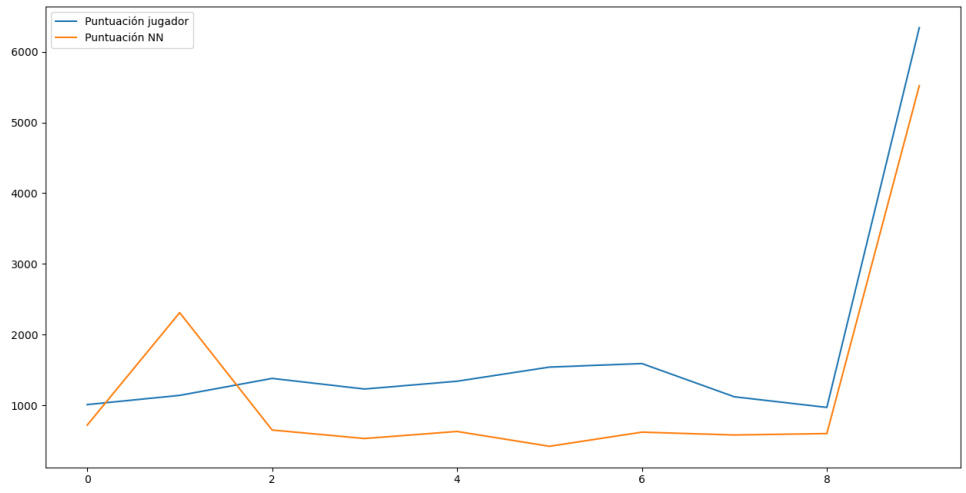

Figura 5.20: Puntuación a lo largo del conjunto de partidas contra un jugador.

A lo largo de las 10 partidas las puntuaciones entre jugador y modelo no tuvieron mucha disparidad, manteniéndose cercanas en casi todas las partidas. El jugador humano fue mejorando su puntuación poco a poco mientras se adaptaba a la forma de jugar del modelo,

mientras que este intentaba hacer lo propio, como se puede ver en la Figura 5.20. Cabe destacar la última partida, donde las puntuaciones se dispararon por la duración de la misma, pero a pesar de la victoria del jugador el modelo mantuvo también una buena puntuación.

En comparación con las otras pruebas la diferencia de puntos entre ganador y perdedor a penas supera los 1000 puntos a lo sumo, véase la Figura 5.21. Las partidas terminaban en los primeros turnos, por lo que la puntuación de ambos era reducida impidiendo que la diferencia de puntos creciese como en los otros casos. Aún así,en la última partida se llegó a una cantidad de puntos elevada y la diferencia entre el modelo y el jugador no aumento por ello. Esto da a entender que el modelo, a pesar de no ganar más de una partida, mantuvo un nivel de juego adecuado.

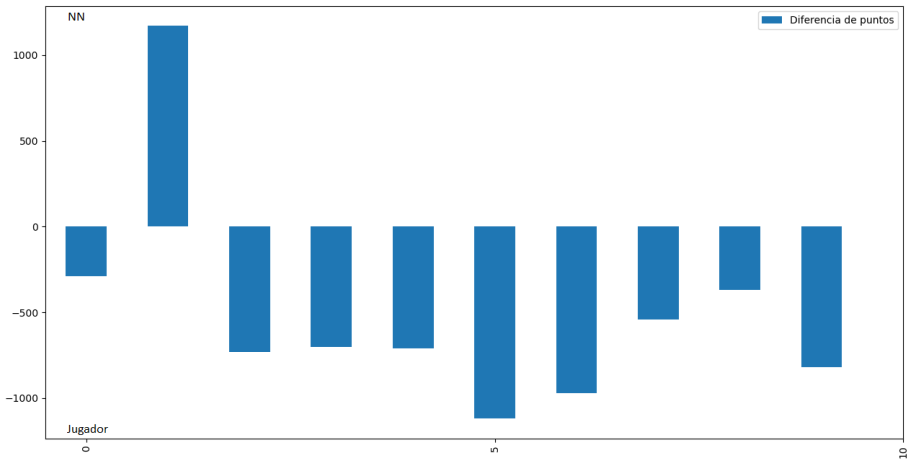

Figura 5.21: Diferencia de puntuación a lo largo del conjunto de partidas contra un jugador.

Finalmente, aunque el modelo consiguió unas buenas puntuaciones tan solo logró una victoria a lo largo de los enfrentamientos como se puede ver en la Figura 5.22. Por los resultados de las pruebas contra el MCTS y el otro modelo cabría esperar que, si se aumentase el número de enfrentamientos contra el jugador, el modelo terminase por adaptarse y mejorar sus resultados.

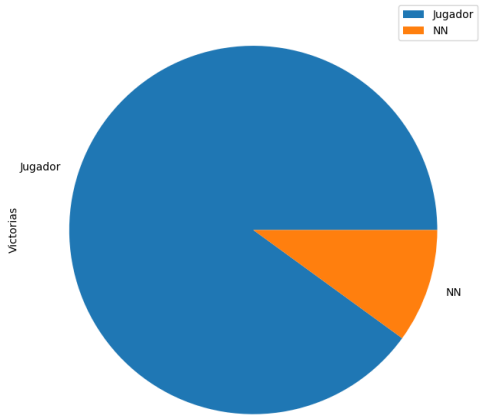

Figura 5.22: Número de victorias en los enfrentamientos contra un nuevo modelo.

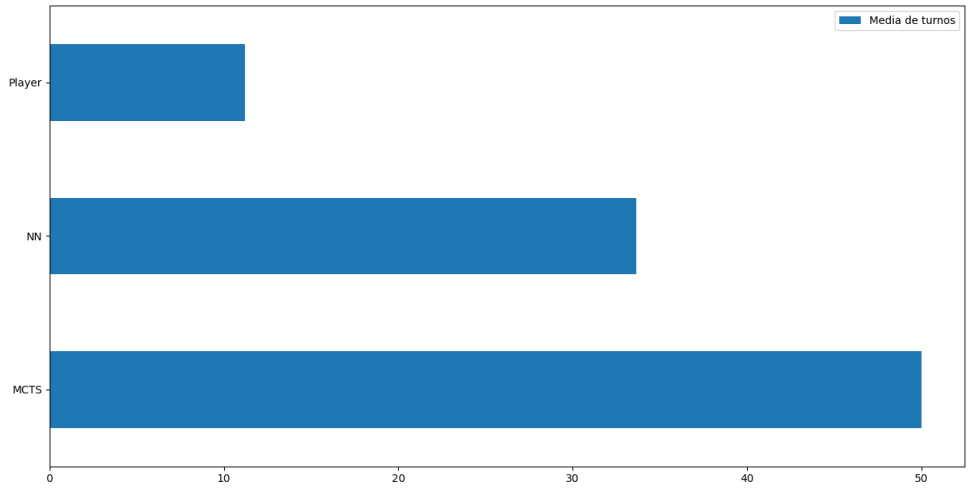

Figura 5.23: Media de número de turnos por partida por oponente.

Como dato interesante sobre la duración de las partidas, en la Figura 5.23 se observa como contra el MCTS ninguna partida terminó antes de los 50 turnos de límite, por lo que se puede suponer que los enfrentamientos tendían a ser igualados a pesar de tener unas puntuaciones con cierta distancia. Por otro lado, se ve que las partidas contra el jugador son las más cortas,

esto es debido a la diferencia de nivel entre jugador y modelo. Y, por último, las partidas entre los dos modelos se encuentran en el medio con una duración media de entre 30 y 40 turnos, que demuestra que a pesar de tener enfrentamientos con cierta igualdad entre ambos, en algunos casos uno de los modelos sobrepasaba al otro con cierta rapidez.

Capítulo 6

Conclusiones y Futuros desarrollos

El presente capítulo se dedica a revisar el desarrollo del Trabajo Fin de Grado, así como concretar las lecciones aprendidas de manera resumida. También se dedica el apartado 6.4, a desgranar posibles líneas de desarrollo futuro que se abren con la consecución de este trabajo.

6.1 Conclusiones del proyecto

Cuando un proyecto finaliza, la primera de las tareas debe de ser verificar el cumplimiento de los objetivos planteados al inicio y el grado de consecución de los mismos. A modo de adelanto, se puede decir que todos los objetivos se han cumplido de manera satisfactoria.

Primeramente, se debe destacar que se ha conseguido desarrollar la biblioteca de apoyo al entrenamiento por aprendizaje por refuerzo planteada al inicio; esta cuenta con una interfaz fácilmente implementable que, además, se ha usado para implementar un ejemplo. Este ejemplo, a su vez, ha servido para realizar pruebas sobre la biblioteca y servirá como pauta para todo aquel desarrollador que desee dar uso de la biblioteca.

Se debe destacar que el desarrollo de la biblioteca ha conseguido implementar la aproximación planteada en el Capítulo 2. Esta es una aproximación nueva que toma como eje central los 3 elementos que se han identificado como clave en todo juego de mesa: jugador, juego y ronda.

Asimismo, la implementación del ejemplo ha servido para comprobar que la estructura basada en los juegos de mesa y usada por la biblioteca se adapta correctamente a otros problemas similares, como es en este caso un videojuego de estrategia.

Con la biblioteca y el ejemplo listos, se realizaron diversas pruebas para comprobar que resultados se pueden conseguir entrenando un modelo con esta biblioteca y cuyo análisis puede verse en detalle en el Capítulo 5.

Por lo tanto, se ha diseñado e implementado una biblioteca que expone un API para ayudar a realizar entrenamiento por aprendizaje automático aprovechando los beneficios de la aplicación de la estructura de los juegos de mesa. Se ha desarrollado un ejemplo, necesario no solo para la realización de las pruebas, sino para demostrar cómo se ha pensado que se deben usar las interfaces de la biblioteca.

Tras la realización de las pruebas y el posterior análisis de los datos recogidos, se puede decir que el modelo entrenado contra el algoritmo MCTS lo ha superado en rendimiento, con mejores puntuaciones y mayor número de victorias, y en tiempo, un turno del MCTS podía llegar a durar 30 minutos mientras que en el caso del modelo era casi instantáneo. Así que se puede decir que este tipo de entrenamiento consigue resultados, como mínimo, tan buenos como el algoritmo escogido para la resolución del problema.

Por otra parte, también se ha comprobado que se puede entrenar un nuevo modelo utilizando a otro como rival consiguiendo nuevamente buenos resultados. Incluso llegando a terminar la partida por conquista de la base del oponente, un caso que no se dio en ninguno de los enfrentamientos realizados contra el MCTS. Esto podría significar que un entrenamiento con la estructura seguida durante las pruebas, enfrentando al modelo primero contra un algoritmo capaz de resolver el problema y luego contra si mismo o un nuevo modelo, puede llegar a conseguir unos resultados excelentes.

Por desgracia, los peores resultados se obtuvieron en la parte final de las pruebas, al enfrentarse contra un jugador humano. El hecho de conseguir al menos una victoria es destacable, aunque por los resultados anteriores podía esperarse un mejor resultado. Aún así la tendencia de los resultados podría indicar que si el modelo siguiese enfrentándose al jugador acabaría por aumentar el número de victorias a su favor.

Es por ello que el trabajo puede ser calificado de éxito al cumplir con todos los objetivos planteados.

6.2 Resultado del seguimiento

Al comienzo del proyecto se realizó una estimación en tiempo y coste, pero a lo largo de la duración del proyecto se han sufrido desviaciones como es habitual. Parte del proyecto se

desarrolló durante el estado de alarma producido por el COVID-19; debido a esto el CESGA limitó las ejecuciones no asociadas a la investigación del virus. Con estas limitaciones el proyecto sufrió desviación en tiempo en el último *sprint*, ya que las ejecuciones de las pruebas llegaron a retrasarse días. Finalmente, se tuvo que ampliar una semana el último *sprint* como caso excepcional por la situación.

Esto se reflejó también en los costes aumentando las horas a pagar al analista en 40 horas. Por esas 40 horas que no se habían tenido en cuanto se pagarían 625 €. En el caso del coste de los recursos materiales, una semana extra de uso significaría un aumento de 53,5 € sobre el coste estimado. Por otro lado, los costes indirectos el aumento sería de 350 €. Esto suma un total de 1.028,5 € por encima del coste total estimado al inicio del proyecto.

6.3 Conclusiones personales

A modo de lecciones aprendidas, de la realización de este proyecto destaca el uso de nuevas tecnologías como Tensowflow con Python muy en boga en los últimos tiempos tanto en el sector académico como empresarial. Así mismo el trabajo ha servido para aplicar y consolidar los conocimientos adquiridos durante el grado como, por ejemplo, los conceptos sobre el aprendizaje automático, en concreto el aprendizaje por refuerzo. De esta forma se refuerzan y consolidan dichos conocimientos tras haber sido llevados a la práctica. También, con el juego de ejemplo, se llevan a la práctica conocimientos sobre desarrollo de videojuegos igualmente obtenidos en el grado, afianzándolos al programar todas las mecánicas del juego desde cero. Con las pruebas, se ha visto en práctica la parte más teórica sobre redes neuronales, sus entrenamientos y sus posibles problemas. Centrándose en evitar el sobre entrenamiento de cualquiera de los modelos utilizados durante las mismas.

Por otro lado, se han aplicado tanto Scrum y Kanban de la mejor manera posible teniendo en cuenta las limitaciones del trabajo, siendo la principal que el equipo de desarrollo esté formado por una sola persona. Aún así, ha servido para conocer mejor la aplicación real de dichos métodos y afianzar los conocimientos que ya se tenían de ellos.

6.4 Líneas de trabajo futuro

Durante el desarrollo del trabajo y a la finalización del mismo se han ido encontrando posibles mejoras tanto para la biblioteca, como para el ejemplo y la realización de pruebas que

se escapaban del alcance del mismo pero que podría ser interesante añadir. A continuación se listan algunas de ellas:

- Buscar más puntos generalizables para aumentar la cantidad de métodos que pueda aportar la biblioteca.
- Realizar un mayor número de pruebas con un jugador humano y comprobar cómo aprende el modelo. Con el objetivo de conseguir un modelo superior al jugador.
- Si se desease continuar con las pruebas contra un jugador sería interesante implementar una interfaz gráfica. Esto ayudaría al jugador a entender mejor el estado del campo de batalla mejorando su forma de jugar.
- Generalizar la clase utilizada para la ejecución, configuración y registro de datos de las pruebas. Esos tres puntos son comunes a cualquier implementación de la biblioteca, si se añade a la propia biblioteca se puede evitar la repetición de la programación de la clase.
- Añadir un sistema de análisis de datos que facilite la selección de un modelo tras la realización de las pruebas. Con los datos extraídos de las pruebas es posible hacer un primer filtro a la hora de seleccionar un modelo entre varios posibles.
- Portar la biblioteca a lenguaje Julia, ya que es un lenguaje en auge en el campo del aprendizaje automático.

Apéndice A

Apéndices

A.1 Estudio tecnológico

A continuación, se detallan las tecnologías utilizadas durante el desarrollo del trabajo, tanto el lenguaje como las bibliotecas utilizadas y las herramientas que lo han facilitado.

A.1.1 Lenguajes

Python

Python[19] es un lenguaje de programación interpretado, con tipado dinámico y multi-paradigma, con soporte tanto para programación orientada a objetos como para imperativa entre otras. Se creó con la idea de hacer código fácilmente legible.

Posee una gran comunidad, siendo uno de los lenguajes de programación más usados a día de hoy[20]. Gracias a eso se tiene acceso a una amplia variedad de bibliotecas que aportan funcionalidades de todo tipo. En particular, en el campo del aprendizaje automático es el lenguaje con mayor apoyo por parte de la comunidad[21].

Por todo esto se decidió utilizarlo, en su versión 3.7.3, para desarrollar la biblioteca, así como el ejemplo y el procesado de los datos obtenidos durante las pruebas.

A.1.2 Bibliotecas

Tensorflow 2.0

Tensorflow[22] es la biblioteca para aprendizaje automático más utilizada en lenguaje Python. Desarrollada por Google pero de código abierto, aporta capacidades para desarrollar y entrenar redes neuronales.

Su API tiene diversas capas de abstracción, utilizando Keras es muy fácil definir un modelo y realizar su entrenamiento.

Actualmente se encuentra en su versión 2.1, versión que se ha utilizado en este proyecto. Se usó para la definición y creación de los modelos entrenados durante las pruebas, así como para la ejecución del propio entrenamiento.

Keras

Keras[23] es una biblioteca para redes neuronales pensada para ser amigable para el usuario. También es de código abierto y actualmente está incluida en Tensorflow 2.0.

Ofrece capacidades para la creación de capas y modelos, así como métodos para el entrenamiento y la evaluación de dichos modelos.

En este proyecto se ha utilizado junto con Tensorflow en su versión 2.2.4. La mayoría de funcionalidades usadas en el proyecto de la biblioteca de Tensorflow pertenecen a Keras.

Numpy

Numpy[24] es una de las bibliotecas más usadas en Python. Al igual que todas las bibliotecas anteriores es un proyecto de código abierto.

Contiene un gran número de métodos para realizar operaciones matemáticas de alto nivel, dando soporte para el manejo de *arrays* y matrices.

Como en casi todo proyecto con Python, Numpy se ha usado para el manejo de los datos en su versión 1.18.

Pandas

Pandas[25] es una biblioteca para el manejo de datos que amplia las funcionalidades de Numpy. Suele ser utilizada para la creación de gráficas con el fin de analizar los datos representados.

Dispone de estructuras de datos, operaciones para su manipulación y métodos para la visualización de dichos datos.

Todas las gráficas y el análisis de los datos de las pruebas del proyecto se han realizado con esta biblioteca. Se ha usado la versión 1.0.4.

A.1.3 Herramientas de apoyo al desarrollo

Visual Studio Code

Visual Studio Code[26] es un editor de código gratuito. Es conocido por su capacidad para adaptarse a cualquier entorno de desarrollo gracias al gran abanico de extensiones que mantiene su comunidad.

Ha sido el editor utilizado para toda la programación realizada durante el proyecto. Para poder utilizarlo con Python de forma adecuada se instaló la extensión oficial que ofrece Microsoft.

Git

Git[27] es una herramienta de control de versiones(desde ahora VCS), es decir registra los cambios de los archivos del proyecto y mantiene un repositorio. Además, permite mantener un repositorio en local y en un servidor, así como gestionar los cambios entre los archivos en ambos. Otra característica importante de Git es la posibilidad de la creación de ramas para separar el flujo de trabajo y unirlo posteriormente. Es la herramienta de este estilo más usada en la actualidad.

Para su uso en este proyecto se utilizó el Gitlab que proporciona la facultad como servidor. Para ello se creó un repositorio y se siguió el flujo de trabajo GitFlow.

GitFlow consiste en la gestión de las ramas del repositorio de una forma concreta y para

ello se requieren las siguientes ramas:

- *Master*. Los cambios en esta rama deben estar listos para pasar a producción.
- *Develop*. Los cambios en esta rama formarán la siguiente versión del proyecto, se incorpora a Release.
- *Release*. Esta rama es el paso anterior a Master, donde se comprueban los cambios antes de incorporarlos.
- *Feature*. Estas ramas surgen de la rama Develop y se reincorporan a ella. Se utilizan para añadir nuevas funcionalidades al proyecto.
- *Hotfix*. Estas ramas se originan en Master y se incorpora a Master y Develop. Se utilizan para arreglar problemas que aparezcan en producción.

Jira

Jira[28] es una aplicación web de gestión de tareas, especializada en el uso de metodologías ágiles. Permite la creación de historias de usuario y tareas, la gestión de *sprints* y la organización de tableros Kanban y Scrum.

Servidores de pruebas

Para la ejecución de las pruebas fue necesario el uso de unos servidores, en concreto los proporcionados por el CESGA.

Para ello se han usado dos programas para poder conectarse a dichos servidores. Primero fue necesario el uso de Forticlient[29], un cliente de conexión VPN, ya que el CESGA no permite conexiones no protegidas. El otro programa fue MobaXterm[30], un cliente SSH y SCP con terminal y explorador de archivos gráfico.

A.2 API de la librería

La biblioteca expone una serie de interfaces que forman el API de la misma. Estas interfaces deben ser implementadas por cada desarrollador que desee utilizarla. A continuación se describe cada interfaz, así como los métodos que deben ser implementados para su uso.

La interfaz "Juego"(*Game*) debe ser implementada por la clase encargada de la gestión de las reglas que se han de seguir para resolver el problema. Se deben implementar los siguientes métodos:

- getBoard() : Board. Devuelve el tablero correspondiente al estado actual.
- getScore() : List<int>. Devuelve una lista con la puntuación actual asociada a cada uno de los jugadores.
- getActions() : List<Action>. Devuelve una lista con las acciones posibles en la situación actual.
- doAction(action: Action) : void. Recibe una acción que debe existir en la lista de acciones posibles y la ejecuta actualizando el estado.

La interfaz "Jugador"(*Player*) debe ser implementada por las diversas clases que vayan a resolver el problema. Se deben implementar los siguientes métodos:

- __init__(player : int) : void. Recibe un número que identificará al jugador.
- takeAction(game : Game) : Game. Recibe el problema, toma una decisión sobre las acciones posibles, la ejecuta y devuelve el problema en su nuevo estado.
- train(rounds: List<Round>) : void. Recibe una lista con las rondas que se han ejecutado desde su último entrenamiento y, si se desea, se realiza el entrenamiento con esos datos.

La interfaz "Ronda"(*Round*) debe ser implementada por una clase que guarda la información necesario de cada ronda que se ha ejecutado. Se debe implementar el siguiente método:

- __init__(score : List<int>) : void. Recibe una lista con las puntuaciones obtenidas durante la ronda por cada uno de los jugadores.

Por último, también es necesario desarrollar una clase que implementa la interfaz "Tablero"(*Board*) encargada de guardar la información del estado en el que se encuentra el problema. Y otra clase, o clases, que implemente la interfaz "Acción"(*Action*) que debe contener la información y los métodos necesarios para modificar el estado del problema. Ambas interfaces dejan al desarrollador la libertad de implementarlas cómo mejor le convenga para el problema a resolver en concreto.

Apéndice B

Glosario de acrónimos

API *Application Programming Interface*

CESGA *Centro de Supercomputación de Galicia*

HU *Historia de Usuario*

IA *Inteligencia Artificial*

MCTS *Monte Carlo Tree Search*

RF *Requisito Funcional*

RNF *Requisito No Funcional*

UML *Unified Modeling Language*

VCS *Version Control System*

Apéndice C

Glosario de términos

Aprendizaje automático Campo de las ciencias de la computación que busca la creación de algoritmos que den lugar a modelos basados en un conjunto de datos que puedan generalizar su comportamiento para un conjunto de datos mayor.

Aprendizaje por refuerzo Aprendizaje automático no supervisado que basa su funcionamiento en la toma de acciones sobre un entorno, recibiendo una recompensa y el nuevo estado del entorno como retroalimentación.

Neurona artificial Unidad de calculo que intenta simular el comportamiento de una neurona real.

Red de neuronas artificiales Modelo computacional formado por un conjunto de neuronas artificiales conectadas entre sí que ante una entrada realiza ciertas operaciones para producir una salida.

Bibliografía

[1] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel *et al.*, "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018.

[2] RedmineUP, "Agile plugin," https://www.redmineup.com/pages/es/plugins/agile, Visitado el 18 de mayo de 2020.

[3] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[4] A. E. Sallab, M. Abdou, E. Perot, and S. Yogamani, "Deep reinforcement learning framework for autonomous driving," *Electronic Imaging*, vol. 2017, no. 19, pp. 70–76, 2017.

[5] S. Mahadevan and J. Connell, "Automatic programming of behavior-based robots using reinforcement learning," *Artificial intelligence*, vol. 55, no. 2-3, pp. 311–365, 1992.

[6] I. Szita, "Reinforcement learning in games," in *Reinforcement learning*. Springer, 2012, pp. 539–577.

[7] S. B. V. D. S. Q. X. Z. X. Ravichandra Addanki, Mohammad Alizadeh, "Understanding & generalizing alphago zero," https://openreview.net/forum?id=rkxtl3C5YX, Visitado el 24 de mayo de 2020.

[8] G. Piatetsky, "Interview with rich sutton, the father of reinforcement learning," https://www.kdnuggets.com/2017/12/interview-rich-sutton-reinforcement-learning.html, Visitado el 24 de mayo de 2020.

[9] J. Kober, J. A. Bagnell, and J. Peters, "Reinforcement learning in robotics: A survey," *The International Journal of Robotics Research*, vol. 32, no. 11, pp. 1238–1274, 2013.

[10] J. P. Morgan, "Machine learning in fx," https://www.jpmorgan.com/global/markets/machine-learning-fx, Visitado el 24 de mayo de 2020.

[11] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, p. 484, 2016.

[12] R. Coulom, "Efficient selectivity and backup operators in monte-carlo tree search," in *International conference on computers and games*. Springer, 2006, pp. 72–83.

[13] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton *et al.*, "Mastering the game of go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[14] I. Ghory, "Reinforcement learning in board games," *Department of Computer Science, University of Bristol, Tech. Rep*, vol. 105, 2004.

[15] R. S. Sutton, "Learning to predict by the methods of temporal differences," *Machine learning*, vol. 3, no. 1, pp. 9–44, 1988.

[16] Scrum.org, "What is scrum?" https://www.scrum.org/resources/what-is-scrum, Visitado el 17 de mayo de 2020.

[17] D. Radigan, "Kanban: Aplicación de la metodología kanban en el desarrollo de software," https://www.atlassian.com/es/agile/kanban, Visitado el 17 de mayo de 2020.

[18] w84death, "Tanks of freedom," https://github.com/w84death/Tanks-of-Freedom, Visitado el 17 de mayo de 2020.

[19] Python.org, "Python," https://www.python.org/, Visitado el 17 de mayo de 2020.

[20] TIOBE, "Tiobe index," https://www.tiobe.com/tiobe-index/, Visitado el 8 de junio de 2020.

[21] T. Elliott, "The state of the octoverse: machine learning," https://github.blog/2019-01-24-the-state-of-the-octoverse-machine-learning/, Visitado el 8 de junio de 2020.

[22] Tensorflow.org, "Tensorflow," https://www.tensorflow.org/, Visitado el 8 de junio de 2020.

[23] Keras.io, "Keras," https://www.keras.io/, Visitado el 8 de junio de 2020.

[24] Numpy.org, "Numpy," https://numpy.org/, Visitado el 8 de junio de 2020.

[25] Pandas, "Pandas," https://pandas.pydata.org/, Visitado el 8 de junio de 2020.

[26] Microsoft, "Visual studio code," https://code.visualstudio.com/, Visitado el 12 de junio de 2020.

[27] S. F. Conservancy, "Git," https://git-scm.com/, Visitado el 12 de junio de 2020.

[28] Atlassian, "Jira," https://www.atlassian.com/es/software/jira, Visitado el 12 de junio de 2020.

[29] Fortinet, "Forticlient," https://www.forticlient.com/, Visitado el 12 de junio de 2020.

[30] Mobatek, "Mobaxterm," https://mobaxterm.mobatek.net/, Visitado el 12 de junio de 2020.